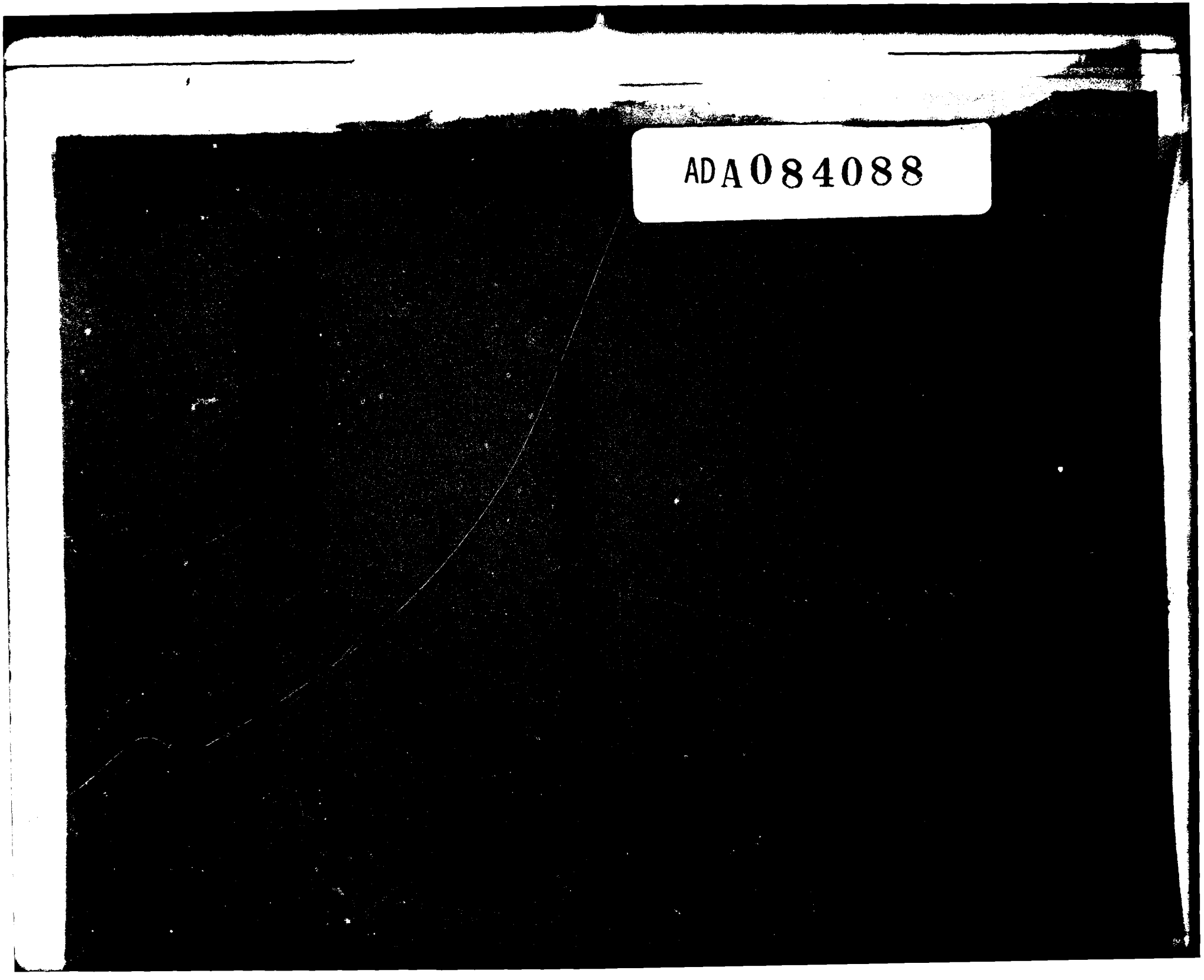
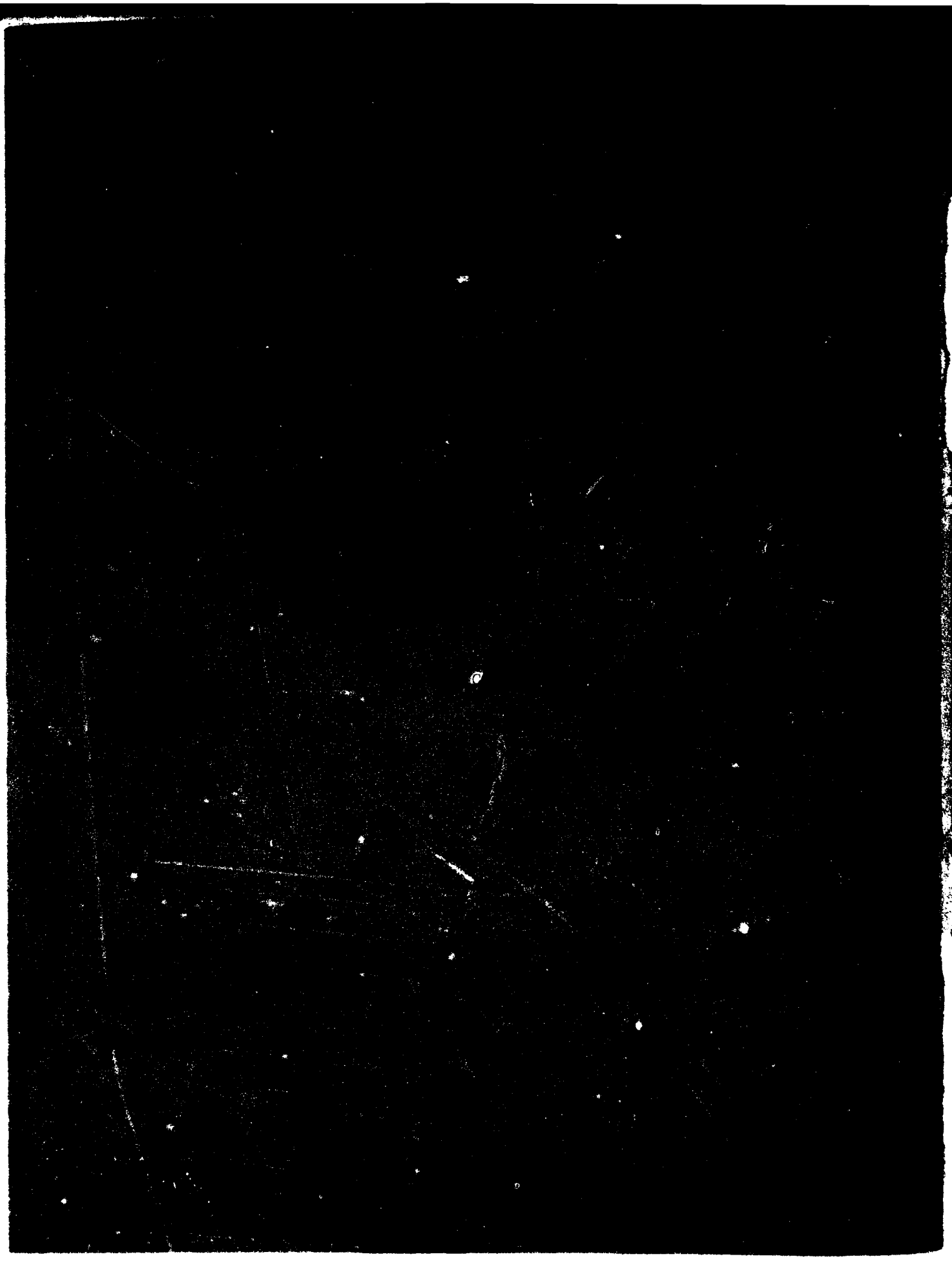
AD-A084 088 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA F/G 9/2
UNIX NSW FRONT END.(U)
MAR 80 R H THOMAS F30602-78-C-0242
UNCLASSIFIED RADC-TR-80-44 NL





UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-80-44	2. GOVT ACCESSION NO.: AD-A084 088
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): UNIX NSW FRONT END	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. Sep 78 — Sep 79
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Robert H. Thomas	8. CONTRACT OR GRANT NUMBER(s): F30602-78-C-0242
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Bolt Beranek and Newman, Inc. 50 Moulton Street Cambridge MA 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F 55812102 21
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISCP) Griffiss AFB NY 13441	12. REPORT DATE: Mar[illegible]80
	13. NUMBER OF PAGES: 118
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Capt George Brooks (ISCP)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

National Software Works (NSW)	Front End
ARPANET	UNIX
Network Operating Systems	Resource Sharing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The UNIX NSW Front End is a user interface to the National Software Works (NSW) System designed to run under the UNIX Operating System on a PDP-11 Computer connected to the ARPANET. This report documents the design for the UNIX NSW Front End and the status of its implementation.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

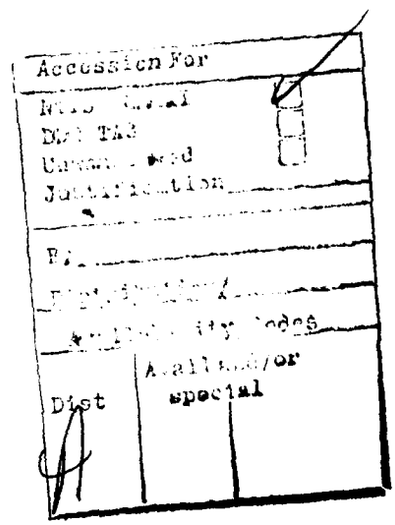
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

'000100

JCB

Contents



EVALUATION

The UNIX NSW Front End Final Report provides a technical perspective for developments in the user-interface front end area as it applies to the distributed heterogeneous computer network known as the National Software Works (NSW). The report identifies and discusses development status, and most importantly, provides recommendations as to which issues require further development in order that continued orderly progress may be made toward NSW Technology Transfer.

This effort applies to TPO R3D, specifically, Project 5581, "Information Sciences Technology", (C^2 Information Processing Thrust) and to Project 2526, Task 07, "National Software Works Technology Demonstration". The results from this development effort will and have been used in formulating the continued technical program under Project 5581.

George E Brooks

GEORGE E. BROOKS, Capt, USAF
Project Engineer

1. Introduction

The NSW Unix Front End project is part of the National Software Works (NSW) program sponsored jointly by the Air Force and ARPA. The goal of the NSW program is to develop a network operating system that provides an effective environment for software production, software configuration control, and software maintenance.

The objective of this contract was to develop a user interface to the National Software Works system for a PDP-11 with the Unix operating system. The interface is called an NSW Front End.

Development of a Unix NSW front End involved several tasks:

- Designing a user command language for the Unix Front End.
- Designing software to implement the Unix Front End functions.
- Modifying the Unix operating system to enable it to support the Front End software.
- Implementing the Unix Front End software.

At the end of the contract the status of these tasks was as follows:

- The user command language has been designed and documented. The appendix to this report documents the design.
- The Unix Front End software has been designed and documented. The software design is also documented in the appendix.
- The modifications to Unix required to support the Front End have been completed and debugged, and are now part of the

standard Unix system used, maintained and distributed by BBN. The modifications included enhancements to the Unix terminal handler, the ARPANET network control program (NCP), and the TELNET protocol implementation. These changes are also documented in the appendix.

- The Unix Front End software is partially complete. The Unix Front End may be used to log into NSW and to invoke most of the functions provided by the NSW system. Section 2 describes the status of the Front End software in more detail.

A contract for a follow-on project to complete and enhance the Unix Front End is expected. The objectives of the contemplated project are threefold:

- To enhance the capabilities of the Unix NSW Front End developed under this contract. This will include completing implementation of the Front End design as documented in the appendix, as well as designing and implementing several new Front End features.

- To install the Unix operating system and the Unix NSW Front End on a PDP-11/45 at Warner-Robins Air Force Base.

- To support and maintain the software that implements the Unix NSW Front End.

At the completion of the follow-on effort the Unix Front End will be an easy to use, reliable, maintainable software product.

There are two parts to the rest of this report. Section 2 describes the current status of the NSW Unix Front End. The appendix documents the design for the Front End command language, the design for the Front End software, and the modifications made to the Unix operating system to support the Front End.

2. Status of the Unix NSW Front End

The Unix NSW Front End is designed to run under the Unix operating system on a DEC PDP-11/45 or PDP-11/70 connected to the ARPANET.

As noted in Section 1, the Front End software is partially complete. It currently runs on the BBN-UNIX host (ARPANET host number 77 (octal)). The Front End was initially implemented and debugged to run against release 3.1 of the NSW system. The current version has been upgraded to run against release 4.1 of NSW, the most recent release of the system.

In addition to the modifications to Unix, implementation of the Front End involved software development activity in two areas.

- Unix MSG.

  The Front End must communicate with NSW software modules on other hosts. To do so it uses MSG, the NSW interprocess communication facility. A partial and largely untested implementation of MSG for Unix, developed under Navy and ARPA funding, was available at the beginning of the current contract. This implementation had to be completed and debugged.

- The Front End software.

  The principal software development activity was to implement the NSW Front End functions. These functions fall into two categories: the user interface, which provides the user with a friendly environment for interacting with NSW by means of NSW commands; and a network interface, which communicates with NSW software modules on other hosts in order to carry out the user's commands.

The status of the software development in these areas at the end of the contract was:

- Unix MSG.

  The MSG implementation is partially complete. Two of the three communication modes supported by MSG, message communication and alarm communication, have been implemented. Implementation of the third mode, direct connections, is not complete. The partial implementation of MSG that we started with was in worse shape than we had thought, and we were forced to spend significantly more effort than we had expected to make it usable. In addition, the current Unix MSG implementation has performance limitations which must be improved before the Front End software can be considered a product.

- The Front End software.

  The Unix Front End currently supports all NSW user commands except those that require the use of MSG direct process to process connections (see above). That is, all user commands except those to start and stop tools are supported (1). In particular, the following commands are supported (see the appendix for descriptions of the various commands): abort, alter, copy, delete, describe, fastout, help, job, login, logout, net, password, quit, rename, semaphore, and show. The NSW commands not currently supported are: use, quit, and reuse. In terms of the two aspects of the Front End, the user interface and the network interface, the user interface is complete and the network interface is partially complete.

We recommend that the following tasks be performed as part of the new contract:

---

1. All of the code required to communicate with and control tools has been implemented and tested. The testing was accomplished by implementing a user TELNET facility in the Front End. (Management of tool connections is essentially identical to managing TELNET connections.) Of course, this code cannot be used for tool interaction until the MSG implementation is complete.

- The Unix MSG should be completed by implementing direct process to process connections.

- The NSW commands for starting and stopping tools on interactive tool bearing hosts should be implemented.

- The performance of the Unix MSG should be improved.

1. Introduction

This document describes the design for a Front End for the National Software Works System. The design presented is for a Front End to run under the UNIX operating system.

The National Software Works (NSW) is a physically distributed computer operating system designed to run on a heterogeneous collection of computers (hosts). For the current implementation of NSW, the computers are linked at the network level by the ARPANET. NSW is not an operating system in the ordinary sense. That is, it is not concerned with directly managing the physical resources (e.g., central processors, memory, i/o channels) of the constituent hosts. Rather, it is a meta-operating system in that it is built upon the operating systems of the constituent hosts and acts to manage the host resources in a way that enables the resources found on the different hosts to be used together in an integrated fashion.

NSW is a mission oriented system intended to facilitate software production. To this end it provides for direct user communication with useful software production aids, called tools. Typical tools include editors, compilers, loaders and test data generators, that are resident on the various hosts. NSW performs two important functions related to tool support. It provides users with a uniform means to access any tool regardless of the host that supports the tool, and it provides both users and tools a uniform way to access any file regardless of the host location of the file. In addition, NSW provides a framework for the

development of "management" tools which can be used by managers to monitor and control the progress of software production projects.

The following briefly reviews the NSW system architecture. More details on the system architecture may be found in [1], [2], [3], [4], [5] and [6].

The principal NSW system components are the Front End, Works Manager, Foreman, and File Package. Users access NSW resources through the Front End component. The Front End mediates between the user, other NSW components, and tools supported under NSW. It is the only NSW component that directly interfaces to the user.

The Front End communicates with other NSW components by means of an inter-host interprocess communication facility called MSG. MSG provides for communication between all NSW components. The resource management and access control functions of NSW are logically centralized in the Works Manager component. The Works Manager is responsible for maintaining a central file catalogue, initiating tool sessions for users, and logging users in and out of the system.

NSW tools run on tool-bearing hosts (TBHs). The Foreman component is part of the TBH software. It is responsible for controlling the execution of NSW tools on the TBH. The Foreman starts tool sessions on its TBH at the request of the Works Manager. Part of its function is to provide an interface between the tool on its host and the rest of NSW. There is an NSW File

Package module on all NSW TBHs and on any other NSW hosts that store NSW files. The role of the File Package is to manage the physical copies of NSW files, to transport files between hosts, and to perform file format and data translations in support of tool execution and other user activity.

1.1 Design Goals

The Front End design we present has two major goals: a well-engineered human interface and high performance. Possibly the performance goal is implied by the user interface goal, but we regard it as sufficiently important to warrant separate mention. The Front End performance measure we consider most critical is responsiveness. Since adequate performance of distributed systems such as NSW has been difficult to achieve in the past, the UNIX Front End should from the outset be designed with this goal in mind.

1.2 Organization of This Document

The purpose of this document is to describe the design for a UNIX NSW Front End. Section 2 describes broadly the functions which an NSW Front End must perform. Section 3 describes the constraints within which the UNIX Front End design must be developed. The user interface that is to be provided by the UNIX Front End is described in Sections 4 and 5. Section 6 describes the approach we plan to take for implementing the Front End. Sections 7, 8, and 9 describe, respectively: changes to be made to the UNIX operating system in order to enable it to support the

Front End design; a proposed change to NSW protocol which will simplify implemenation of the Front End; and some enhancements to UNIX MSG that are required to support the Front End implementation. Finally, Section 10 discusses an issue regarding the implementation of some of the NSW management tools that must be resolved before the UNIX Front End can be completed.

2. Front End Functions

An NSW Front End (FE) performs several distinct functions. This section discusses the principal FE functions.

2.1 Command Interpretation

The FE supports the NSW command language, the means by which a user interacts with NSW. Command interpretation involves parsing user typein and initiating the NSW system operations required to satisfy valid commands.

2.2 Interaction with NSW System Components

In order to satisfy user requests the FE interacts with other NSW system components. These interactions are governed by a set of NSW protocol "scenarios". These protocol scenarios specify the system components involved in each NSW system operation and the communication that occurs among them to perform the operations (see [7],[8]). The intercomponent interactions themselves are supported by means of MSG. For the current set of protocol scenarios, the FE interacts only with the Works Manager and Foreman components.

2.3 Tool Session Control

The FE provides the user means to control tool execution. This involves the initiation and termination of tool sessions as well as supporting direct communication with the tool. Initiation and termination require interaction with other system components. The result of tool initiation is the creation of a communication path between the Front End and an active tool (and

its Foreman). Usually the communication path is a TELNET connection, although it need not be. A user may have multiple tool sessions active at the same time. To support multiple tools, the FE must provide means by which a user may switch his attention back and forth among the various active tools and between the tools and the NSW command interpreter.

2.4 Terminal Control

The user and the Front End interact by means of the user's terminal. The FE exerts control over a number of basic terminal handling functions, such as the manner in which various non-printing "control characters" are represented when echoed and output, the input characters that may cause "program interrupts", and so forth. The FE makes use of the UNIX terminal handling modules to control these functions. In addition, the FE provides the means by which a user may edit his typein before the FE acts on it.

3. Constraints on the Front End Design

The FE design for the FE we present is constrained in several ways. The factors which constrain our design are discussed in this section.

3.1 Functions Supported by NSW

Unlike the other major system components (i.e., MSG, Foreman, File Package) there is no System Subsystem Specification for NSW Front Ends (1). There are, however, some "de facto" specifications. These specifications derive principally from the existence of a version of the NSW system. In the current system, there are a collection of procedures in other system components (primarily the Works Manager, but including the Foreman) which are callable by the FE. These FE-callable procedures, in a sense, define the NSW system functionality in that they represent the system operations a user can invoke through an FE. The UNIX FE will provide means for users to invoke these NSW system operations.

3.2 Existing Intercomponent Protocols

As noted in Section 2.2, there is a set of protocol scenarios which define how the various system components are to interact to implement NSW system operations. The operation of

---

1. There is a document [9] which was written to be a specification for a minimal function NSW FE. However, changes in the NSW system design have rendered that document obsolete. The document does have limited utility as a guideline for general Front End issues.

the UNIX FE must, of course, conform to these protocol scenarios. Work in the area of NSW performance improvements is likely to result in changes to certain of the protocol scenarios. The UNIX FE implementation will be structured so that it can be changed relatively easily to participate in any modified protocol scenarios.

The protocol scenarios prescribe patterns of communication among NSW system components. Another protocol called NSWTP (for NSW transaction protocol) prescribes conventions for the contents of messages exchanged as part of the protocol scenarios. NSWTP is principally concerned with issues such as how to distinguish between a request message and a response message, and where in a message the function requested and its parameters are to be found. In Section 8 we propose a minor change to NSWTP that is intended to facilitate FE operation.

An FE's communication with other NSW system components is by means of MSG. The UNIX FE must, of course, use the MSG conventions when it engages in the various NSW protocol scenarios. A partial implementation of MSG for UNIX currently exists; the principal deficiency is that MSG direct connections are not supported due to limitations in the UNIX NCP (shortly to be corrected). As part of this project, the UNIX MSG implementation will be completed. The UNIX FE itself will not include the MSG functionality but rather will invoke the communications services of UNIX MSG.

Although the NSW system design does not require it, at present all communication between NSW tools and FEs is by means of MSG direct connections. With one exception (the NLS tool) these connections are TELNET connections. This requires that the UNIX FE adhere to the TELNET protocol in its interactions with (most) NSW tools.

3.3 Current NSW TENEX FE

A less stringent constraint on the UNIX FE design is the current existence of a TENEX NSW Front End. The TENEX Front End was implemented primarily to serve as a debugging aid; however, it was for a long time, and still is, the only NSW FE. Apart from equivalent functionality, no relationship between the TENEX FE and UNIX FE is required. However, we believe that it is desirable for the UNIX FE user interface to be compatible, or nearly compatible, with that of the existing TENEX FE.

3.4 UNIX Operating System

The impact of the UNIX operating system on the FE design is mostly in the area of the structure of the implementation. Limitations of the UNIX terminal handler may have a minor impact on the functions that can be supported at the user interface. Section 7 outlines modifications to UNIX that are intended to minimize this impact. This section discusses some of the constraints imposed by UNIX on the FE implementation.

The UNIX FE will be implemented as a collection of UNIX processes (see Section 6). The UNIX operating system does not

support shared memory between processes. Consequently, any data sharing that is required between processes that implement the FE must be accomplished by some other means. We plan to use shared files and interprocess pipe transmissions in situations that require cooperation. Refer to Section 6.4.2 for further discussion of this point.

UNIX imposes a site-dependent limitation on the number of active file descriptors allowed per process. Typically, file I/O occurs with one read file descriptor and one write file descriptor, although read-write file I/O with a single file descriptor is possible. Two-way inter-process pipes require two file descriptors; the terminal requires three file descriptors. The number of file descriptors available affects the FE implementation by imposing a limit on the number of files and pipes usable per process. There is no limitation, apart from those generated by system resources and system loading, on the number of processes that may be allocated to a user job.

3.5 Future Evolution of the NSW

Whenever possible, the FE design should be flexible enough to permit easy implementation of projected future modifications to NSW. Certain proposed NSW system changes, such as tool-chaining (i.e., the sequential use of more than one tool in the same tool workspace), will require more extensive interactions between the FE and the Foreman. For example, as part of the tool chaining feature the FE and Foreman will support commands to manipulate tool workspaces and to move files between the NSW file system and the workspace.

FE processing of NSW protocol scenarios will be implemented in a table-driven form, so that any changes to the scenarios may be easily incorporated into the FE.

At present the specification of the NSW command language is largely up to the FE designer. The language is likely to change in response to user feedback and the need for further standardization in the future. Therefore, provision for easy modification of the command syntax is part of the FE design; for example, the parser for the command language will be table-driven.

4. New Front End Features

Three major features, not currently supported by the TENEX NSW Front End, are planned for the UNIX Front End. These features are:

- Immediate Command Return Mode.
- Command Procedures.
- User Profile.

Only the first will be supported by the initial implementation of the UNIX FE.

4.1 Immediate Command Return Node

In the present FE after a user completely specifies an NSW command the FE retains control until the commmand is completed. That is, the user's terminal is disabled for further input until the NSW completely processes the command. Because most commands involve interactions with system components resident on remote hosts which may take a long time to complete, the users often find themselves in situations where they can do nothing but wait. The situation is made worse by the fact that there is no way for a user to interrupt or abort a command after it is initiated. This has resulted in a high level of user frustration with the NSW system.

The UNIX FE is designed to operate in two "command return" modes: "deferred return" mode for which it retains control until the command is completed; and "immediate return" mode whereby control is returned to the user immediately after a protocol

scenario for the command is initiated rather than after the scenario completes.

The immediate return mode permits the user to initiate other commands while the protocol scenario for a commmand is being performed. When the scenario completes the user is notified that the command has completed. The user may display any output produced as the result of the command execution when, and if, he wishes by an explicit or implicit "display" command.

The user may specify the command return mode he wishes by an FE command. When the user profile feature is implemented, the preferred return mode may be stored in the profile and set automatically at user login.

When the FE is operating in immediate return mode commands are, in effect, performed in the "background". Operation in this mode involves a style of interaction between the user and the FE which requires communication regarding on-going commands. The FE must inform the user when such a command completes or needs help; and, the user must inform the FE when ready to see the output produced by a completed commmand, when he wants an on-going command to be canceled, or when ready to supply help for an on-going command.

To support this style of interaction, when a command is initiated by the user the FE will assign it a numeric name and report the name to the user; e.g.,

```
NSW: delete a.b.c
     Delete [7] initiated.
NSW:
```

When the command completes the FE will notify the user; e.g.,

```
     Delete [7] competed.
NSW:
```

or

```
     Show Files [12] completed; output ready.
NSW:
```

The user can use the "display" command to see output produced by completed commands; e.g.,

```
NSW: display (output for command) 12
```

When help is needed by the FE to complete a command the FE will notify the user; e.g.,

```
     Help for Delete [7] required.
NSW:
```

The user may supply help at his convenience by means of the "help-reply" command; e.g.,

```
NSW: help-reply (for command) 7
       .
       .
       .
```

When the FE is operating in immediate return mode the potential for unsequenced execution of on-going commands exists. That is, if the user has initiated several commands the FE could in principle execute them in sequence, in parallel, or in any order. For some sets of commands the order of execution would not matter, but for others, such as ones that include file delete and rename commands, the outcome would depend on the order of execution. For the initial implementation of the UNIX FE all

commands will be executed in sequence. Later implementations may support various levels of unsequenced command execution.

4.2 Command Procedures

When implemented, the command procedure feature will permit a user to define "composite" commands which consist of a sequence of NSW FE commands and which are to be executed as a single command.

The definition for a composite command or command procedure will be a text file stored in the NSW file system. When a command procedure is invoked by the user, the corresponding definition file will be retrieved from the NSW file system and interpreted by the FE. Definitions for command procedures may include conditional command execution as well as transfers of control. In addition, command procedures will be able to be passed arguments when invoked, and will, as part of their execution, be able to gather input from the user's terminal.

The command procedure feature is not fully designed at this time.

4.3 User Profile

At some point in the future the NSW will support the concept of a user profile. The user profile will enable a user to have the system store information about his preferred use of NSW and to have it set various system usage parameters for him automatically when he logs in.

When implemented, the user profile feature will include means to permit users to enter, modify and display the settings in their profiles. The feature will require support from the Works Manager as well as from the Front End.

At present the user profile capability for NSW is not designed. However, when the capability is designed the UNIX FE will be enhanced to support it.

5. User Interface

The UNIX FE is designed to provide a "friendly" interface to NSW that is well engineered to the needs of NSW users. The user interface is a reasonably complex entity. For purposes of documenting it, it is convenient to divide the user interface into smaller parts. These parts are:

- abstract syntax.

  This defines the general syntactic structure of the FE command language.

- command editing conventions.

  These are the means by which a user may correct his typein before it is acted upon by the FE.

- help facilities.

  These are the means by which a user may request assistance, such as prompts, as he interacts with the FE.

- parsing rules.

  These define how the FE acts to interpret user typein.

- command return conventions.

  These define the way the FE returns control to the user after a command is entered.

- FE commands.

  The specific FE commands invoke the various functions supported by the FE.

Together these comprise the user interface provided by the FE. This section discusses each of them in turn after first describing the two basic modes of FE operation.

5.1 Modes of Operation

At any given time during an NSW session the user's FE operates in one of two modes: exec mode or tool mode. When it is in exec mode, the FE is receptive to commands from the user. These commands request either NSW system operations (e.g., run a tool) or local FE actions (e.g., switch into tool mode). In exec mode the FE processes all user typein and tries to accumulate a complete command. When it has a complete command, the FE "executes" it either by initiating an NSW protocol exchange scenario with other system components or by performing the requested action locally. When the FE is in tool mode user typein is treated as input for a tool and is forwarded by the FE to the appropriate tool.

5.2 Abstract Syntax for the FE Command Language

This section specifies an "abstract syntax" for the FE command language. By abstract syntax we mean a definition of the general structure of the command language without a specification of language details such as the specific commands it includes or the particular characters used as punctuation within command strings.

It is useful to identify an abstract syntax for the FE because it helps simplify the description of the user interface. The complete specific FE syntax is defined by the abstract syntax along with a list of the commands supported by the FE, the

punctuation rules the FE uses, and other special interaction conventions it uses (e.g., editing, help). The FE commands are described in Section 5.10, punctuation is described in Section 5.8, and other conventions are described in Sections 5.3, 5.4, and 5.5.

The particular abstract syntax we propose for the UNIX FE is motivated by the current TENEX NSW FE. It is presented below using a variant of BNF. We use the convention that terminal items (i.e., those which do not appear on the left side of productions) begin with lower case letters and non-terminal items (i.e., those that appear on the left side of productions, and therefore never appear in strings that are in the language) begin with upper case letters.

The abstract syntax for the FE command language is:

```
<Complete-command> = <Command><cc>

<Command> = <Command/params> |
            <Command/params><mc><Command-mods>

<Command/params> = <Command-spec> |
            <Command-spec><tc><Params>

<Command-spec> = <command-verb> |
            <command-verb><tc><Command-qualification>

<Command-qualification> = <qual-item> |
            <qual-item><tc><Command-qualification>

<Params> = <param> |
            <param><tc><Params>

<Command-mods> = <Mods/params> |
            <Mods/params><mc><Command-mods>

<Mods/params> = <Mod-spec> |
            <Mod-spec><tc><Params>
```

```
<Mod-spec> = <mod-item> |
             <mod-item><tc><Mod-qualification>

<Mod-qualification> = <qual-item> |
             <qual-item><tc><Mod-qualification>
```

All of the terminal items are strings of one or more characters as follows:

Item	Type	Meaning
<command-verb>	string	The name of a command.
<qual-item>	string	Qualification for a command or command modifier.
<param>	string	A parameter.
<mod-item>	string	The name of a command modifier.
<cc>	special char or string of special chars	Signals end of complete command.
<tc>	"	Terminates string.
<mc>	"	Signals beginning of command modification.

If one considers an alphabet consisting of strings (S) and the special characters <cc>, <mc> and <tc>, the FE abstract syntax generates the regular expression language:

S {<tc>S}* {<mc>S{<tc>S}*}* <cc>

(where * means "any number of occurrences of, including none") or, equivalently:

```
S {<tc>S}* {<tc>S}* {<mc>S {<tc>S}* {<tc>S}*}* <cc>
1    2        3          4    5        6
```

where 1 is the command name, 2 is a list of command qualifications, 3 is a list of command parameters, and 4 through 6 is a list of command modifications, where 4 is a command modifier, 5 is a list of qualifications to the command modifier, and 6 is a list of parameters for the modifier.

Some examples may help to clarify the abstract syntax.

Consider:

```
<command-verb> = {use, show, copy, delete}
<qual-item> = {files, node}
<param> = {NSW Filespecs, Tool Names, Access Types}
<cc> = CR LF
<tc> = SPACE
```

With these specific additions the following commands are in the language generated by the abstract syntax:

```
use teco
show files any foreman...source
copy foreman.encap.source new.foreman.encap.source
delete dispatcher.sav
show node
```

These commands are quite similar to NSW commands supported by the current TENEX FE. The sections that follows will show that the language defined by the abstract syntax together with the editing and parsing rules is almost identical to the current FE language.

From the point of view of the command language itself, it makes little difference which terminal items are <qual-item>s and which are <param>s since the set of commmand strings generated by the abstract syntax are identical. For example, it makes little difference to the user in terms of what are legal FE commands whether FILES and NODE are <qual-item>s or <param>s. However, there is a difference from the point of view of how the FE supports the command language. For example, since the <qual-item>s that are legal in any context are known and relatively few, the FE can support recognition and completion for them (See Section 5.4), whereas it cannot for <param>s since all

the legal values a <param> can assume in a particular context cannot easily be known by the FE.

The TENEX FE does not currently include the notion of command modification as represented by the <Command-mods> item. The idea is that command modification would be used whenever a user wished to specify more precisely the action he wants the system to take. Command modification would never be required for any command but could be added at the user's option. For example, a user may want to be able to specify in more detail the information to be displayed by certain SHOW commands. Consider the following additions to the syntax from the example above:

```
<mod-item> = {tool-rights, projects, nodes, set-semaphores}
<mc> = COMMA CR LF
<qual-item> = {..., sessions}
```

The following commands are generated by the abstract syntax:

```
show files any bbn...sources,
set-semaphores

show node,
tool-rights

show sessions,
nodes BBN.Schantz MCA.Fanuef,
projects CCN
```

Here the first command would display only files satisfying the filespec whose semaphores were set. The second command would display only the tool rights for the user's node. The third command would display active user sessions of the indicated nodes and projects. (None of these functions is currently supported by the Works Manager.)

The commands to be supported by the UNIX FE are listed in Section 5.10. For each <command-verb> the valid <qual-item>s and <mod-item>s are listed.

5.3 FE Command Editing

The command editing functions provide a means for a user to change his typein before the FE acts upon it.

The following functions will be supported:

- Abort Input.

  If the "abort input" character is typed while the FE is gathering a <Complete-command>, the partially completed command is deleted in its entirety, and the FE signals the user that it is ready for a new command by outputting the FE command prompt (see Section 5.7).

- Erase Character.

  The "erase character" character causes the FE to delete the last character typed and to print the character that was deleted. This function works until the beginning of the word (string) is reached. At that point the FE signals the user that no more characters remain to be deleted.

- Erase Word.

  The "erase word" character causes the FE to delete either the current partially completed word or if the user is at the at beginning of a word, the previous word. This function works until the beginning of a command is reached. At that point the FE signals the user that no more words remain to be deleted.

- Retype Command.

  The "retype command" character causes the FE to output the user's (corrected) typein. If there is no typein the FE signals the user that there are no characters to be retyped.

5.4 FE Help Facilities

A user may request information from the FE regarding its use as well as the use of the NSW in several ways. These are:

- By specific command.

  The FE will support HELP and DESCRIBE commands. HELP will print a short message that contains key information regarding operation of the FE (such as special function characters) as well as information on how to obtain more specific assistance. DESCRIBE will print a description of its argument. Neither of these functions is currently supported by the TENEX FE.

- By requesting a prompt.

  At the user's request the FE will prompt for the next field in a command. The user requests the prompt by typing a special prompt request character (e.g., ESC). The FE will respond by printing a prompt string that indicates the type of input it expects (e.g., "(Access type)").

- By requesting the current options.

  At any point in typein the user may ask the FE for the options available to him. He does this by typing a special option request character (e.g., question mark). The action taken by the FE depends upon the position at which the request occurs within the partially specified command. If the position is at the beginning of a <command-verb>, <qual-item> or <mod-item>, the FE types out the list of valid <command-verb>s, <qual-item>s or <mod-item>s. If it is within a <command-verb>, <qual-item> or <mod-item> it types out the list of possible <command-verb>s, <qual-item>s or <mod-item>s that begin with the string the user has partially typed. If it is at the beginning of or within a parameter, the FE prints the type of parameter expected. In all cases the FE resumes its string gathering operation after outputting the options.

- By requesting recognition and completion.

  The user may request the FE to recognize and complete a <command-verb>, <qual-item> or <mod-item>. He does this by typing a special recognition/completion request character (e.g., ESC). The FE will respond by trying to recognize the partially specified string. If the partially specified string unambiguously identifies an item, the FE will recognize it and complete it by outputting its remainder. If the string does not, the FE will signal the user. In either case the FE resumes its string gathering operation.

5.5 FE Parsing Rules

The FE parsing rules define the actions taken by the FE when a user types <cc>, <tc> and <mc>.

One of the goals in formulating the parsing rules was to have the FE provide a "friendly" interface to the user in the sense that the FE should be forgiving rather than vindictive when the user makes an error. For example, whenever feasible the FE should assist the user when he makes an error in entering a command (such as typing <cc> before he has completely specified a command), rather than admonish him by aborting a partially specified command and outputting an error message.

The parsing rules designed for the UNIX FE are the following:

- <tc> - String terminator.

  The FE checks the string terminated by the <tc>.

  If a <command-verb>, <mod-item>, or <qual-item> is expected, the string must be one of the known verbs or items; otherwise the string is an error. If a parameter is expected, the FE tries to verify that the parameter is of the type expected: e.g., filespec, access type, integer, scope, arbitrary string. Of course, it can only check filespecs for syntactic correctness.

  If the terminated string is correct, then the FE attempts to gather the next string. If it is incorrect, then the FE outputs an error indicator and a prompt, and tries to gather the string again.

- <cc> - Command terminator.

  If the previous string was not terminated by a <tc>, the FE first treats the <cc> as a <tc>.

If the string was terminated by a <tc> or if the unterminated string was determined to be acceptable, the FE checks to see if the typein so far completely specifies a command. If it does, the FE asks the user for confirmation prior to performing the command. The user may respond in one of two ways: he may instruct the FE to go ahead and "execute" the command, by typing a second <cc>; or he may use any of the editing characters (including the abort input character) to edit the command.

If the command is incomplete, the FE treats the <cc> as a <tc> (if necessary) followed by a user request for a prompt (i.e., the prompt request character) and prompts the user for the next string.

- <mc> - Command modification indicator.

  If necessary the FE first treats the <mc> as a <tc>.

  Then it checks to see if complete command (i.e., <Command/params>) has been specified. If so, the FE begins to collect command modifications (i.e., <Command-mods>). If a <Command/params> has not been specified, the FE treats the <mc> as a <tc> (if necessary) followed by a user request for a prompt.

5.6 Command Return Modes and Command Names

Section 4.1 described two conventions for returning control to the user after a <Complete-command> is entered and confirmed. For deferred return, the FE retains control and terminal input is ignored until the command completes. For immediate return, the FE returns control to the user as soon as it initiates the actions required to execute the command, and the user may enter new commands while the command proceeds to completion.

The UNIX FE will support both modes of operation. Initially the FE will be in deferred command return mode. After logging in, the user may, if he desires, specify by command the command return mode to be used.

As discussed in Section 4.1, immediate command return mode requires that commands be named so that the FE and user can refer to various on-going commands. The names for commands will be numbers, and they will be assigned by the FE and reported to the user when commands are initiated. The FE will support several commands for dealing with on-going commands while operating in immediate command return mode. The "display" command will cause the FE to print the output resulting from the execution of a specified command. The "help-reply" command can be used to supply help for a specified command. The "abort" command can be used to cancel an on-going command. Of course, the outcome of the abort command is somewhat probabilistic since there is no way of guaranteeing that the command to be cancelled has not already completed or progressed to a point where it cannot be cancelled.

5.7 Miscellaneous

This section describes aspects of the user interface which do not fall neatly into the above five sections.

- FE Command Prompt.

  The FE will have a "prompt" which is used when it is in exec mode to signal the user that it is ready to accept the next command. The command prompt is output in a number of situations, such as upon return to exec mode from tool mode and upon completion of processing for an abort input character. The command prompt for the current TENEX FE is the string "NSW:". At present we see no reason to change it for the UNIX FE.

- Switching Modes.

  The TENEX FE recognizes a special character (CNTL-N, for NSW) as a request to switch from tool mode to exec mode. It uses another special character (CNTL-U, for use) to switch from exec mode to tool mode. The UNIX FE will recognize the same special character (CNTL-N) when in tool mode as a request to switch modes. It will provide a <command-verb> for switching from exec mode to tool mode. (It may also support a special character for this purpose. However, in general, we feel that the number of special characters in the FE command language should be minimized.)

- Query mode input.

  Certain NSW commands require a large number of parameters. The NET command for importing, exporting, and transporting files is a good example. It seems unreasonable to expect any user to remember all of the command parameters required, and the order in which they must be specified for such commands. For these commands the FE will enter into a query mode after the command is partially specified (e.g., a <command-verb> and perhaps a <qual-item> have been specified), whereby it will prompt the user for the required parameters. After all of the parameters have been entered, and before command execution begins, the user will be given an opportunity to confirm or abort the command.

- Quote Character.

  The UNIX FE will support a special quotation character to allow a user to enter special function characters (e.g. <cc>, "delete character" character) as parts of strings rather than to request their special functions.

- User Signal.

  Sections 5.4 and 5.5 mentioned several situations where the UNIX FE will signal the user (e.g., an "erase character" character has been typed but there are no characters remaining to be deleted). The FE will have a special output signal to alert the user in these situations. We have chosen the character CNTL-G (BELL) for this signal.

- Flush Output.

  The Front End will support a "flush output" function. This will enable a user to cause output intended for his terminal to be discarded without causing the operation being performed by the FE to be aborted.

- Interrupt FE.

  When the FE is in exec mode it can be "interrupted" by typing a special interrupt FE character. This causes the FE to output the NSW prompt and to begin gathering a new command. If the FE was operating in deferred command return mode the on-going command, if any, is allowed to continue execution as if in immediate command return mode. If desired, the user may cancel it by using the "abort" command.

- Login Command.

When the UNIX FE is started for a user it will output an NSW/FE herald followed by its command prompt. Since at this point the user has not yet logged in, he will be able to use only a limited number of commands, including a LOGIN command to login (see Section 5.10). This differs slightly from the TENEX FE which does not support a command for login but rather runs in a special mode when it is started where it prompts the user to supply login information.

- TELNET Functions.

When the FE is in tool mode a user will be able to cause the FE to send certain TELNET control characters, such as the TELNET "erase character" character and the TELNET "erase line character", by typing special predefined function characters. In addition, a quote character will be supported to allow the user to cause the FE to send any of the predefined function characters to the remote tool.

- Invoking the FE from UNIX.

There are two possibilities here. One is to allow the user to invoke the FE by typing a command, such as "nsw", to the UNIX shell (command language interpreter). The other is to, in effect, replace the standard UNIX shell with the FE so that when a user activates his terminal he finds himself talking to the FE rather than to the shell. The first approach would be appropriate for UNIX hosts that support other applications in

addition to the NSW FE, and the second is more appropriate for hosts that are dedicated to the FE function. The UNIX FE will be implemented so that it can be invoked in either way.

5.8 Special Character Assignments

The following are the special character assignments for the UNIX FE:

Function	Assignment
<cc>	CR LF (= carriage return followed by line feed)
<tc>	SPACE, TAB
<mc>	COMMA CR LF
abort input	CNTL-X
erase character	DEL
erase word	CNTL-W
retype	CNTL-R
request prompt	ESC
request options	?
recognize and complete	ESC
quote	CNTL-V
return to exec mode	CNTL-N
interrupt FE	CNTL-X
signal to user	CNTL-G (Output from FE to user)

5.9 Example User Interactions

The examples in this section are chosen to illustrate the nature of the interactions to be supported by the UNIX FE. The examples are the five commands from the example in Section 5.2. Output generated by the FE is underlined; user typein is not underlined. The examples are annotated for purposes of explanation; the annotations would not, of course, appear in actual user/FE interactions.

Example 1.

Command: use teco

Interaction:

us(ESC)e (ESC) (tool named) teco ! [Confirm] !
1 2 3 4 5 6 7

Notes:

1. user requests recognition and completion.
2. FE completes "use".
3. user requests prompt.
4. FE outputs prompt.
5. user types <cc> (= CR LF); FE echoes ! for <cc>.
6. FE requests confirmation from user.
7. user confirms with <cc>; FE echoes !.

Example 2.

Command: show files any foreman...src

Interaction:

```
sho (ESC) (item) fi(ESC)les (ESC) (access type) (CNTL-R)
   1 2      3        4   5    6      7              8
sho files any (ESC) (filespec) foreman...src : [Confirm] (CNTL-X)
9             10     11                      12   13       14
```

Notes:

1. user types <tc> (= SPACE); FE recognizes "show" command.
2. user requests prompt.
3. FE outputs prompt.
4. user requests recognition and completion.
5. FE completes "files".
6. user requests prompt.
7. FE outputs prompt.
8. user requests retype.
9. FE retypes command line.
10. user requests prompt.
11. FE outputs prompt.
12. user types <cc>; FE echoes !.
13. FE requests confirmation.
14. user aborts input (and command).

Example 3.

Command: copy foreman.src new.foreman.src

Interaction:

```
copy ?
     1
     filespec for source file
     2
copy foreman.src ! (to file named) new.foreman.src ! [Confirm] !
2                3  4                              5  6         7
```

Notes:

1. user requests help.
2. FE outputs help.
3. user types <cc> before command fully specified; FE echoes !.
4. FE treats <cc> as <tc> followed by ESC and outputs prompt.
5. user types <cc>; FE echoes !.
6. FE requests confirmation.
7. user confirms with <cc>; FE echoes !.

Example 4.

Command: delete dispatcher.sav

Interaction:

```
del cisp(CNTL-W)dis(CNTL-R)
   1       2         3
del dispatcher.sev ! [Confirm] (DEL)\v(DEL)\eav ! [Confirm] !
4                  5  6         7              8  9         10
```

Notes:

1. FE recognizes "delete" command.
2. user erases word.
3. user requests retype.
4. FE retypes command line.

5. user types <cc>; FE echoes !.

6. FE requests confirmation.

7. user notices error in filespec and erases characters.

8. after correcting filespec, user types <cc>.

9. FE requests confirmation.

10. user confirms with <cc>.

Example 5.

Command: show node

Interaction:

```
show ?
     1
     node
     files
     scopes
show nos(DEL)\sd(CNTL-R)
2         3  4      5
show nod(ESC)e ! [Confirm] !
6         7  8 9    10    11
```

Notes:

1. user requests options.

2. FE outputs options.

3. user erases character.

4. FE prints character deleted.

5. user requests retype.

6. FE retypes command line.

7. user requests recognition and completion.

8. FE completes "node"

9. user types <cc>; FE echoes !.

10. FE requests confirmation.

11. user confirms with <cc>; FE echoes !.

5.10 UNIX FE Commands

This section lists the commands to be supported by the initial implementation of the UNIX FE. The list includes all commands currently supported by the TENEX FE, as well as a few additional commands.

- ABORT command.

  Purpose: Cancels an NSW command which has been issued.

  Syntax:

  <command-verb> = abort

  <qual-item>s = none

  <mod-item>s = none

  parameters: Command number.

  Command including prompts:

  NSW: abort (command) CommandNumber

  Notes:

  The command number is assigned by the FE when the command is issued, and it is made known to the user at that time.

- ALTER command.

  Purpose: Alters scopes of NSW files.

  Syntax:

  <command-verb> = alter

  <qual-item>s = add, drop

<mod-item>s = none

parameters: Access type (copy, delete, enter or all) followed by a list of scopes to be altered.

Command including prompts:

NSW: alter (add or drop) add (access type) ScopeList

- COPY command.

Purpose: Makes a copy of an NSW file.

Syntax:

<command-verb> = copy

<qual-item>s = none

<mod-items> = none

parameters: Filespec for the file to be copied; Entryname for the copy to be made.

Command including prompts:

NSW: copy (from file named) Filespec (to file named) Entryname

- DELETE command.

Purpose: Deletes an NSW file.

Syntax:

<command-verb> = delete

<qual-item>s = none

<mod-item>s = none

parameters: a list of (i.e., one or more) Filespecs for the files to be deleted.

Command including prompts:

NSW: delete (files named) Filespec1 ... FilespecN

Notes:

The FE will type out the complete file name corresponding to each of the filespecs and the user will be asked to confirm or abort each delete operation.

- DESCRIBE command.

Purpose: Prints a short description of NSW terms and commands.

Syntax:

<command-verb> = describe

<qual-item>s = none

<mod-item>s = none

parameters: Command name or NSW term.

Command including prompts:

NSW: describe (item) CommandName

- DISPLAY command.

Purpose: Displays output resulting for a completed command.

Syntax:

<command-verb> = display

<qual-item>s = none

<mod-item>s = none

parameters: Command number.

Command including prompts:

NSW: display (output for command) CommandNumber

Notes:

The command number is assigned by the FE when the command is issued, and it is made known to the user at that time.

- FASTOUT command.

  Purpose: Aborts all interactive tool session and logs out of NSW.

  Syntax:

  <command-verb> = fastout

  <qual-item>s = none

  <mod-item>s = none

  parameters: none

  Command including prompts:

  NSW: fastout

- HELP command.

  Purpose: Displays a "help" string to guide user.

  Syntax:

  <command-verb> = help

  <qual-item>s = none

  <mod-item>s = none

  parameters: none

  Command including prompts:

  NSW: help

- HELP-REPLY command.

  Purpose: Used to supply a reply to a help call made by another NSW component.

  Syntax:

  <command-verb> = help-reply

  <qual-item>s = none

<mod-item>s = none

parameters: Command number.

Command including prompts:

NSW: help-reply (for command) CommandNumber

Notes:

The command number is assigned by the FE when the command is issued, and it is made known to the user at that time. The help-reply command prints the message associated with the help call before it begins to gather the reply from the user.

- JOB command.

Purpose: Determines status of batch job.

Syntax:

<command-verb> = job

<qual-item>s = none

<mod-item>s = none

parameters: Job id

Command including prompts:

NSW: job (status of job) JobID

- LOGIN command.

Purpose: Creates an NSW user session.

Syntax:

<command-verb> = login

<qual-item>s = none

<mod-item>s = none

parameters: Project name (PName); Node name (NName); Node password.

Command including prompts:

NSW: login (project) PName (node) NName (password) Password

Notes:

The node password is not echoed by the front end.

- LOGOUT command.

Purpose: Logs out of NSW.

Syntax:

<command-verb> = logout

<qual-item>s = none

<mod-item>s = move

parameters: none for unmodified command; for the <mod-item> move the parameters are: Project name (PName), Node name (NName), and password.

Command including prompts:

NSW: logout

or

NSW: logout,
move (project) PName (node) NName (password) Password

Notes:

1. If there are any active tool sessions the logout will fail.

2. If the move <mod-item> is used, a new NSW session will be created for the specified node after first logging out the old session.

- MOVELOG command.

Purpose: Logs out of NSW and creates a new user session without breaking the connection to NSW.

Syntax:

<command-verb> = movelog

<qual-item>s = none

<mod-item>s = none

parameters: Project name (PName); Node name (NName); Node password.

Command including prompts:

NSW: movelog (project) PName (node) NName (password) Password

Notes:

Identical to a logout command with the move <mod-item>. Movelog is retained for compatibility with the TENEX FE.

- NET command.

Purpose: Moves files into and out of the NSW file system.

Syntax:

<command-verb> = net

<qual-item>s = import, export, transport

<mod-item>s = none

parameters: After the <qual-item> is specified the FE enters query input mode wherein the user is prompted for the parameters required to specify the external (non-NSW) file(s). For "net import" the parameters collected in query mode are the filespec for the NSW destination file and the information necessary to access the external file. For "net export" they are the filespec of the NSW source file and the information necessary to store the destination external file. For "net transport" the parameters collected in query mode are the information necessary to access the source external file and store the destination external file.

Command including prompts:

NSW: net (direction) import
[query mode dialogue]

or

NSW: net (direction) export
[query mode dialogue]

or

NSW: net (direction) transport
[query mode dialogue]

- PASSWORD Command.

Purpose: Changes password for current node.

Syntax:

<command-verb> = password

<qual-item>s = none

<mod-item>s = none

parameters: Old password (Pwd1); New password (Pwd2); New password (Pwd3).

Command including prompts:

NSW: password (current password) Pwd1 (new password) Pwd2 (repeated) Pwd3

Notes:

1. The passwords are not echoed by the FE.

2. The new password is required twice to minimize the chance that the user has mistyped it.

- QUIT command.

Purpose: Ends an active tool.

Syntax:

<command-verb> = quit

<qual-item>s = abort, terminate

<mod-item>s = none

parameters: Tool name

Command including prompts:

NSW: quit (action) abort (tool name) ToolName

or

NSW: quit (action) terminate (tool name) ToolName

- RENAME Command.

Purpose: Changes the name of an NSW file.

Syntax:

<command-verb> = rename

<qual-item>s = none

<mod-items>s = none

parameters: Filespec for file to be renamed; Entryname for new name for the file.

Command including prompts:

NSW: rename (file named) Filespec (to have name) Entryname

- REUSE Command.

Purpose: Switches from exec mode into tool mode, connecting the user to the specified tool.

Syntax:

<command-verb> = reuse

<qual-item>s = none

<mod-item>s = none

parameters: Tool name

Command including prompts:

NSW: reuse (tool name) ToolName

Notes:

1. This command is equivalent to the CNTL-U function in the TENEX FE.
2. If tool name is null, the last active tool is assumed to be the tool name parameter.

- SEMAPHORE command.

Purpose: Controls semaphore on an NSW file.

Syntax:

<command-verb> = semaphore

<qual-item>s = read, set, unset

<mod-item>s = none

parameters: Filespec

Command including prompts:

NSW: semaphore (action) read (file name) Filespec

or

NSW: semaphore (action) set (file name) Filespec

or

NSW: semaphore (action) unset (file name) Filespec

- SHOW command.

Purpose: Prints information about files, node, or scopes.

Syntax:

<command-verb> = show

<qual-item>s = files, node, scopes

<mod-item>s = none

parameters: none for "show node"; for "show scopes" the parameter is an access type; for "show files" the parameters are an access type followed by filespecs for the files of interest.

Command including prompts:

NSW: show files (access type) AccessType (file) Filespec

or

NSW: show node

or

NSW: show scopes (access type) AccessType

- USE command.

Purpose: Opens a connection to an interactive tool and switches from exec mode to tool mode.

Syntax:

<command-verb> = use

<qual-item>s = none

<mod-item>s = none

parameters: Tool name

Command including prompts:

NSW: use (tool name) ToolName

6. Implementation Approach

This section sketches the approach we plan for the Front End implementation. The UNIX FE will be implemented in the C programming language.

6.1 Decomposition into Tasks

The FE will be built from three logically distinct tasks: a user interface task, a protocol interface task, and a tool interface task. These tasks are described in this section. Section 6.4 discusses how UNIX processes will be used to implement these tasks.

6.1.1 User Interface Task

The main responsibility of the user interface task, or user task for short, is to handle all interactions with the NSW user. Its direct interface to the user is through the UNIX terminal handler. The user task implements all control character functions, performs command editing, and parses the user's input. It maintains the user's exec mode/tool mode context. When in exec mode it decides whether a command requires interaction with other system components (through the protocol task) or can be executed directly. When the FE is in tool mode, it forwards the user's typein to the correct tool.

6.1.2 Protocol Interface Task

The protocol interface task, or protocol task for short, is responsible for initiating and participating in NSW protocol scenarios. It initiates protocol scenarios at the request of the

user interface task as required to satisfy user commands, it responds to requests from other NSW system components to engage in protocol scenarios, and it initiates protocol scenarios when certain exceptional conditions are detected (e.g., the autologout scenario is invoked neither by user request nor by remote request, but rather by the FE when necessary). The protocol task engages in protocol scenarios by sending and receiving MSG messages and alarms, and by opening and closing MSG direct connections. Thus the protocol task is activated by requests from the user task, by (MSG) messages from other system components, and by the occurrence of exceptional events. The protocol task treats each on-going protocol scenario as a set of separate transactions and maintains status information regarding the progress of each scenario. Thus it is capable of engaging in several scenarios simultaneously. When a protocol scenario completes or when user intervention is required to continue with a scenario (e.g., to disambiguate an NSW filespec) the protocol task interacts with the user task. In these cases the protocol task signals either the completion of a scenario (possibly along with some text to be displayed to the user) or the need for more information from the user.

6.1.3 Tool Interface Task

The tool interface task, or tool task for short, is responsible for the maintenance of all Front End-to-tool communication. As previously noted, FE-tool communication will for the most part be via the TELNET protocol. When a user is

interacting with a tool, the connection to that tool is said to be active; at other times, it is said to be suspended. The tool task suspends and activates connections, buffers output (from remote tools) for suspended connections, and handles TELNET or other ARPANET protocol obligations required to support tool communication.

For the initial implementation of the UNIX FE, the tool task will support only TELNET communication with tools.

6.2 Communication Between Tasks

Operation of the FE will, of course, require communication between the three tasks. The principal communication patterns are described below.

- User Interface - Protocol Interface Communication

  The protocol task and user task interact with each other to initiate and carry out NSW protocol scenarios. After a user has completely specified a command that requires a protocol scenario, the user task communicates with the protocol task to cause it to initiate the corresponding scenario. Certain protocol scenarios may require further user interaction to be completed. Text to be displayed to the user is sent to the user task from the protocol task, and any user responses are sent back to the protocol task by the user task.

- User Interface - Tool Interface Communication

  The user and tool tasks communicate with each other to handle user/interactions with tools. Changes in context between exec mode and tool mode require communication between these tasks. Changes in tool connection status initiated by the user must be communicated to the tool task. For example, activating a suspended tool connection, suspending an activated one, or setting an echo option involves this communication. The information transmitted between the tasks is typically an indicator of the tool-managing action to be taken and a connection identifier (e.g., the UNIX file descriptor corresponding to the tool connection involved).

- Protocol Interface - Tool Interface Communication

When the protocol scenario to initiate a tool session has completed, the protocol task passes the tool task the information necessary to support communication with the remote tool. The tool task manages the tool connection until it is to be closed, at which time the protocol task assumes responsibility for it. The tool task and protocol task must communicate in order to pass responsibility for the connection back and forth.

6.3 Principal FE Data Bases

The FE uses a number of internal data bases or tables to support its operation. This section describes the principal ones.

6.3.1 Protocol Scenario Tables

As noted in Section 6.1.2, the protocol task is capable of engaging in multiple independent NSW protocol scenarios simultaneously. This capability is supported by several related data bases: the protocol scenario definition table, the active scenario table, and the MSG transaction table. There is an entry for each instance of a protocol scenario in the active scenario table. The entry contains all the information required by the protocol task to perform the particular scenario instance. It holds a pointer to an entry in the protocol scenario definition table; that entry in the protocol scenario definition table identifies the particular protocol scenario (e.g., RUNTOOL, DELETE FILE) and defines the FE's protocol obligations for it in terms of a sequence of steps (e.g., send a generic message, receive a specific message). A complete NSW protocol scenario involves the execution of one or more MSG operations. The

information associated with MSG operations is stored in transaction blocks. This information includes that required to perform the MSG operations as well as information relevant to the protocol scenario that is extracted from data transferred by the operations. As a particular protocol scenario instance progresses, a collection (list) of transaction blocks associated with it grows. The active scenario table entry for a particular scenario instance points to the list of related transaction blocks, as well as to the current transaction block.

6.3.1.1 Protocol Scenario Definition Table

The protocol scenario definition table (or protocol table for short) stores in tabular form the prescribed steps of each NSW protocol scenario the FE may be called upon to participate in. Consequently, protocol changes are easily absorbed by the FE by merely making alterations to the protocol scenario definition table. The active scenario table entry corresponding to a scenario instance points to a particular step of some entry in the protocol table. That step describes the state of the protocol scenario instance.

Each step in a protocol scenario definition includes:

- an operation (e.g., Send primitive)
- an operation modifier
- an argument (e.g., the name of a procedure)
- an argument modifier

Operations are either instructions to the protocol manager (e.g., a conditional or unconditional "goto" operation to modify the position of the pointer in the active scenario table entry to

point to some other protocol step), or they directly correspond to MSG primitives being issued and completed. The procedure name names a local internal FE procedure that is to be called when the step is executed.

6.3.1.2 Active Scenario Table

The active scenario table holds information that records the progress of each active instance of a protocol scenario from start to finish. It is used by the protocol task to track multiple simultaneous scenarios and to enforce the sequencing of protocol scenarios where required.

For each active protocol scenario, the following information is stored in the active scenario table:

- scenario identifier.
  The scenario identifier serves to uniquely identify the scenario instance to the FE. It is identical to the FE-assigned transaction identifier, if one exists; if there is no FE-assigned transaction identifier, the foreign transaction identifier or null idenfifier is converted to an appropriate FE-compatible form. (See Section 8 for further discussion of scenario identifiers.)

- pointers to previous and next scenarios.
  These pointers are used to link scenarios that must be executed in sequence. The pointers point to other entries in the active scenario table.

- pointer to entry in protocol scenario definition table.
  This identifies the particular scenario in progress.

- pointer to the current step in protocol scenario definition table entry.

- pointer to current transaction block.
  Provides reference to transaction block associated with current protocol step.

6.3.1.3 Transaction Table

The structure of a transaction block is a function of the MSG primitive to which it corresponds. The following information is stored in every transaction block, regardless of the MSG primitive involved:

- pointer to the active scenario table.
  This identifes the protocol scenario instance for which the MSG primitive was executed.
- the corresponding MSG primitive.
- primitive completed flag.
  This flag is set to indicate that MSG has completed the primitive.
- pointers to previous and next transaction blocks, if any, that are part of the protocol scenario.
- event handle returned by MSG.
  After the primitive has been issued, MSG returns an event handle by which to identify the pending event. The event handle is later used to determine whether the pending event has completed or not. (Refer to the UNIX MSG User Manual for details.)
- disposition.
  Disposition field used in MSG primitive calls.
- signal used.
  Signal used in MSG primitive calls.
- process name.
  MSG primitive operations have source and destination process names. This name is the name of the destination process.
- type, tid.
  These are the first two fields of an MSG message in NSWTP format. They are message type and transaction identifier, respectively.
- pointer to an MSG message buffer (send or receive, as appropriate).

The remaining information in a transaction block is dependent on the MSG primitive associated with the transaction. The remaining information maintained for each primitive is:

- SendGeneric.
  The name of the remote procedure called and its arguments are stored. The timeout parameter and boolean wait flag (qwait) are always stored.

- SendSpecific.
  For SendSpecific calls which initiate protocol scenarios, the name of the remote procedure called and its arguments are stored. The timeout parameter and special handling flag are always stored. For tool-termination scenarios, accounting information is also stored.

- ReceiveSpecific.
  For ReceiveSpecific calls which initiate protocol scenarios, the name of the local procedure called and its arguments are stored. The timeout parameter and special handling flag are always stored. For a Help call, the help code (which identifies the kind of help sought) is stored. For tool-initiation scenarios, tool-related parameters are stored. For tool-termination scenarios, accounting information, reason for termination, and termination scenario type are also stored.

- SendAlarm.
  The alarm code is stored.

- OpenConn and CloseConn.
  The connection identifier, connection type, and connection handle returned by MSG (a UNIX file descriptor) associated with the connection are stored.

With the exception of pointers, event handles, and the UNIX file descriptors for open connections, note that the information stored in the transaction blocks are either arguments to MSG primitives or components of MSG messages.

6.3.2 Active Session Table

The active session table holds data that describes the state of the user's NSW session as perceived by the FE. It holds the user's project name, node name, and user ID. In addition, it is used to keep a record of the status of user commands which have not been completed. This command information is used in part to

support the immediate command return mode feature discussed in Section 4.1. It is important to have this information readily accessible to the user task (since the user may at any time query the FE for the status of outstanding commands).

For each outstanding, or active, command, the following information is stored:

- command.
  The particular command being performed.

- scenario identifier.
  The identifier for the protocol scenario instance, if any, corresponding to the command.

- command identifier.
  The command identifier is a number assigned by the FE which is used to identify outstanding commands to the user.

- command status.
  The status indicates whether the command has been "requested" (of the protocol task), "initiated" (by the protocol task), or "completed".

- pointer to previous and next commands.
  These are used to link commands which must be executed in sequence; they are pointers to other command entries in the active session table.

- parameters.
  Parameters stored.

6.3.3 Active Tool Table

The active tool table contains all FE information relevant to handling tools. For each active tool, the following information is stored:

- generic tool name.
  During each user session on NSW, a number of tools are available for use by the user. The user may determine which tools are available (by querying the WM) and may invoke their use. Tools are invoked by generic tool name, e.g., TECO or SOS.

- tool instance name.
  When a tool is invoked according to its generic tool name, an instance of that tool is allocated to the user. Each tool instance has a tool instance name associated with it; this name is assigned by the FE and contains the generic tool name as a substring. For example, a representative tool instance name is "3TECO".

- tool address.
  The MSG process address for the tool Foreman process.

- tool identifier.
  The tool id is a WM-determined integer that uniquely identifies a tool instance.

- tool connection handle.
  The connection handle for the connection to the tool that is returned by MSG (i.e., a UNIX file descriptor for the connection).

- connection management data.
  This data represents the current state of the tool connection. Elements of the vector are: connection type, connection identifier (the preceding two are arguments from FE-OPENCONN), a "1nd saved" flag, character count of buffered output, pointer to the output buffer, and an "in use" flag.

The FE maintains a pointer to the active tool table entry for the most recently activated tool. When the FE is in tool mode this is the entry for the currently active tool, otherwise this is the entry for the tool used last. This pointer is the "current tool" parameter. It is used to select a default tool when the user omits the tool name parameter for the "reuse" command.

6.3.4 Terminal Input and Parsing Tables

An input buffer and several tables are used to support the processing of terminal input from the user when the FE is in exec mode. They are described below.

6.3.4.1 Terminal Input Buffer

The user task maintains a terminal buffer to hold exec mode user typein until a string has been verified to be a complete and valid command. This buffer is manipulated by user command-editing functions.

6.3.4.2 Grammar Tree

The grammar tree stores the FE command language in a tabular form. Each node of the tree stores user prompt strings, the type of input expected next (e.g., <qual-item>, <param>), and (where appropriate) further information to enable the FE to query the user for elements of the command.

6.3.4.3 Parse Tree

The parse tree is a data structure built as the user's typein is parsed. It contains "tokens" for the syntactic items (e.g., <command-verb>s, <qual-item>s, etc.) that have been recognized in the terminal input buffer. For each token in the parse tree there is a pointer to the corresponding position in the terminal input buffer. There is, in addition, a pointer to the grammar tree to indicate the current state of the "parse" of the user's input.

6.3.5 Buffers

In addition to the tables described above, the FE maintains a number of buffers.

6.3.5.1 Message Buffers

The protocol task maintains message buffers for outgoing and incoming MSG messages. Buffer space for messages is allocated and deallocated as needed. Since the FE at all times has a ReceiveSpecific primitive outstanding, there is always be at least one receive buffer allocated.

6.3.5.2 Terminal Output Buffers

All terminal output originating in the protocol task is sent to the user task where it is buffered prior to printing. For each buffer, the following information is stored:

- scenario identifier.
- command identifier.
- sequence number.
  This number enables the user task to distinguish between buffered strings having the same scenario identifier.
- display only/Help reply flag.
  This flag specifies whether the string should be displayed with or without expecting a user response.
- the buffered ASCII string.

6.3.5.3 Tool Buffers

For each tool, there is a tool buffer which is used to hold terminal output from remote tools prior to printing on the user's terminal. Each tool buffer consists of header information and space to hold the output for the terminal. The tool buffers are maintained by the tool task.

For each tool buffer, the following information is stored:

- associated active tool table entry.

- the number of characters buffered (held in active tool table entry)
- pointer to last character written into buffer (held in active tool table entry)
- the buffered data itself

6.4 Tasks and UNIX Processes

We plan to use separate UNIX processes to implement the three tasks described above. The user task will be realized by a user process, the tool task will be realized by a tool process, and the protocol task will be realized by a protocol process. The mapping of tasks onto UNIX processes will be "approximate" in that some tasks will be implemented across process boundaries. As examples: when the FE is in tool mode, the tool process will accept user input; and, when a tool is started, the protocol process will be responsible for entering a large portion of the tool data kept in the active tool table. The former function was specified as part of the user task, and the latter function was specified as part of the tool task.

The particular realization of the tasks by processes (and across process boundaries) was chosen to minimize communication between the processes and to separate functions which must occur concurrently into separate processes.

The discussion of the protocol task noted that protocol scenarios may be initiated by user requests to the protocol task, by remote calls to the FE, or by the occurrence of internal FE conditions whose handling requires protocol scenarios (see Section 6.1.2). In principle, these events may occur in any

order. The FE will also support user type-ahead and an immediate command return mode. It follows from these considerations that the protocol task must perform many functions independently of the user task. Hence, the decision to implement the protocol task and user task by two UNIX processes.

Since there is a very limited degree of concurrent operation required among the user and tool tasks, the need to realize them by separate UNIX processes is less clear. However, we believe that partitioning the functions of the user task and the tool task into separate processes will facilitate implementation.

6.4.1 Process Structure

The three processes will be hierarchically structured. The user process will create the protocol and tool processes as its inferiors.

6.4.2 Interprocess Communication

UNIX interprocess pipes will be used for communication between the protocol process and the user process, and between the user process and tool process. Since communication between the protocol process and the tool process is expected to be relatively infrequent and of short duration (see Section 6.2), there will be no direct pipe between them. Instead, the communication between them will be through the user process.

6.4.3 Interprocess Protocol

All communication between the user process and protocol process is related to protocol scenarios. The user process transmits MSG messages in NSWTP format to the protocol process either to initiate scenarios or to supply help for on-going ones. The protocol process transmits text strings to the user process that are to be displayed to the user and, in some cases, parameters to the user process. When a protocol scenario has completed, the protocol process will notify the user process; after modifying its tables, the user process will instruct the protocol process to discard information relevant to the scenario from its tables. Since there may be multiple simultaneous on-going protocol scenarios, scenario identifiers are required in the messages exchanged between the user and protocol process.

Communication between the user process and the tool process is always related to tools and therefore includes the index for the active tool table entry for the tool. Communication from the user process to the tool process is typically to "begin" a new tool (i.e., to perform any table management actions required before the tool connection can be used), to "finish" a tool being terminated (i.e., to perform any table management actions required to clean up before a tool connection can be closed in an orderly fashion), to activate an existing tool connection, or to perform some connection protocol function, such as setting an echo option. The tool process sends the user process acknowledgments and informs it when a tool has been suspended (as the result of a control-N from the user).

In general, all interprocess communication is structured on a request-acknowledgment basis. That is, one process requests another process to perform a given task; the second process returns a message that acknowledges either the completion of the requested task or the receipt of the request.

A interprocess request includes the following information:

- instruction (e.g., discard a protocol scenario)
- identifier (e.g., scenario identifier or active tool table entry)
- parameters (optional)
  The number, type, and format of parameters, if there are any, are dependent on the instruction.

An acknowledgment includes the following:

- acknowledgment verb
- instruction being acknowledged.
- identifier in request being acknowledged.

6.5 Modular Decomposition of the Processes

The structure of the FE processes is described in this section. All three processes are event driven and share the same basic structure: a set of procedures (to implement either protocol-, user-, or tool-handling functions) which are called by routines that interface to the events that drive the process.

6.5.1 The Protocol Interface Process

The driving events for the protocol interface process are input from the user process pipe and the completion of MSG operations. When an event occurs, the process executes the same main loop regardless of the event. The loop is described below according to the sequence of steps taken.

If there is pipe input from the user process, the pipe interface module processes it (see Section 6.5.1.3). The input may cause a new protocol scenario to be initiated, a new step in an existing protocol scenario to be initiated or a completed scenario to be deleted from the active scenario table. If there is input from MSG (signalled by the completion of an MSG primitive), the MSG interface module processes it (see Section 6.5.1.2). Next, a check is made to see whether any exceptional conditions exist that require the FE to initiate a protocol scenario; if so, the Initiate-Scenario procedure is called (see Section 6.5.1.1). The procedure Modify-ProtocolTable is then invoked to advance any of the protocol scenarios that are ready to be advanced as the result of the event(s) that occurred (see Section 6.5.1.1). Finally, the Issue-Primitive procedure is called to issue MSG primitives that are ready to be issued as the result of the event(s) that occurred (see Section 6.5.1.1).

The protocol interface process is built from the following major functional entities: protocol manager, MSG interface module, pipe interface module, and tool table handler.

6.5.1.1 Protocol Manager

The protocol manager is a collection of procedures which implement various protocol handling tasks. These procedures are called by the protocol process as it executes its main loop, as well as by the MSG interface module, the pipe interface module, and other procedures in the protocol manager.

A protocol scenario instance is described by an entry in the active scenario table and a series of scenario steps (a set of linked transaction blocks). Thus, there is an entry in the active scenario table for each instance of a protocol scenario that has been initiated, and there is a transaction block associated with each protocol scenario step that has been initiated. MSG Receive operations are exceptions in that the scenario and scenario step corresponding to a Receive are determined and initiated (if necessary) when the Receive completes.

Initiating a scenario is accomplished by creating an entry in the active scenario table and setting a pointer in it to an entry in the protocol definition table. Initiating a scenario step causes a transaction block for the step being initiated to be allocated and filled. If the protocol step being initiated involves sending a MSG message, then initiating the step will also require that the message be assembled and made ready to send. It is important to note that initiating a step is logically distinct from issuing the primitive corresponding to the step. This is the case because the constraints imposed by scenario sequencing may require the primitive to be issued at a later time than that of the initiation of the scenario step. Thus, in neither case (initiating a scenario or initiating a scenario step) is a MSG primitive issued; the primitive is issued later by a call to the Issue-Primitive procedure.

The procedures will now be described.

- Initiate-Scenario

  This procedure creates an entry in the active scenario table, given a scenario identifier and a pointer to an entry in the protocol definition table. Based on sequencing requirements for the scenario, the newly created entry may be linked with the entries for previously initiated scenarios.

- Initiate-Step

  This procedure initiates a protocol scenario step. It allocates and fills a transaction block for the scenario step. If the scenario step involves sending an MSG message, procedure Initiate-Message is called to incorporate message-related data and pointers into the information structures.

- Issue-Primitive

  This procedure scans the active scenario table, examining each entry. If a protocol scenario has completed, it notifies the user process. If a protocol scenario has been initiated, but the protocol step has not been initiated (see discussion above), Issue-Primitive initiates the scenario step by calling Initiate-Step. The final step of Issue-Primitive is to check whether an MSG primitive can be issued; if so, it is issued.

The next group of procedures deals specifically with MSG message handling.

- Initiate-Message

  This procedure prepares an MSG message for transmission. It stores message parameters in the transaction block and links the transaction block with the message buffer. On input from the user process, the message is assumed to exist in NSWTP form in a MSG send buffer. If no message exists, this procedure first calls Assemble-Message. Initiate-Message is called by procedure Initiate-Step.

- Assemble-Message

  This procedure is called by Initiate-Message when the protocol process must initiate a scenario unilaterally in response to the occurence of an exceptional event (e.g., the AUTOLOGOUT scenario). Given a pointer to an entry in the active scenario table, Assemble-Message will assemble the

appropriate MSG message in a MSG send buffer. It returns a pointer to the message buffer, a message length, and the corresponding command sequencing information.

- Receive-Message

  This procedure handles MSG messages from other system components when they are received. First it attempts to find an entry in the active scenario table corresponding to the message. If none can be found, the message represents the beginning of a protocol scenario and Initiate-Scenario is invoked to create the new scenario. Next, a transaction block is allocated and filled from the message and the "completed" flag in the transaction block is set so that the scenario can be advanced. Receive-Message examines the contents of the MSG message to determine how the message should be handled. Possible handling includes: transmission of an ASCII string to the user process, storing arguments into the transaction block, or calling the tool table handler to modify information in the active tool table.

The final group of procedures modifies the tables which store the protocol scenario information.

- Modify-ProtocolTable

  Recall from above that entries in the protocol scenario definition table is composed of a series of four-element entries, where each entry corresponds to a protocol step in a protocol scenario or to an instruction to the protocol manager. Procedure Modify-ProtocolTable handles all instructions in the protocol table which are for the protocol manager, i.e., those operations in the table which do not correspond to MSG primitive operations. It is called by Receive-Message to verify the protocol validity of a received message, and it is called after a MSG pending event has completed. In the latter case, the scenario is advanced to the next step of the definition in the protocol scenario definition table by Modify-ProtocolTable; any operations in the table which may additionally alter the pointer's position are also executed.

- Flush-Scenario

  Given a scenario identifier as input, this procedure discards all data and pointers for the scenario from the process tables. It is called by the pipe interface module.

- Send-UserProcess

  This procedure is called by Receive-Message when information must be transmitted to the user process. Based on its input arguments, it extracts from the MSG receive buffer certain information (this usually includes an ASCII string to be displayed to the user) and sends it to the user process.

6.5.1.2 MSG Interface Module

This module is used by the protocol process to deal with MSG. It processes MSG communication from remote processes and monitors the disposition of MSG pending events. When MSG communication to the FE is detected, the protocol process calls the MSG primitive RequestSignal to determine the MSG operation (pending event) that completed.

If the completed event was a MSG Receive primitive, the protocol process immediately issues a new ReceiveSpecific primitive and then calls the procedure Receive-Message.

If the completed event was some other operation, the MSG interface module determines the scenario involved and sets the "completed" flag in the appropriate transaction block. For MSG OpenConn and CloseConn primitives, the tool table handler is then called to add or delete tool-related information from the active tool table.

6.5.1.3 Pipe Interface Module

The pipe interface module handles pipe communication from the user process to the protocol process.

The user process sends two kinds of messages to the protocol process: requests to remove (flush) scenarios from the active

scenario table, and requests that result in the transmission of MSG messages to other system components. If the user process requests that a scenario be flushed, the pipe interface module calls the procedure Flush-Scenario in the protocol manager. Requests to send MSG messages contain the message in NSWTP format.

When a request from the user process requires the transmission of an MSG message, the pipe interface module allocates a buffer and loads the message into it. Three types of messages may be sent: messages that initiate scenarios (e.g., calls on WM procedures), replies to HELP calls, and replies to FE-PREDISPLAY calls. The first type involves initiating a step in a new protocol scenario, whereas the replies involve initiating steps in existing protocol scenarios.

6.5.1.4 Tool Table Handler

To minimize interprocess communication, the active tool table is accessible directly to the protocol process, as well as to the tool process and to the user process. The protocol process, by means of the tool table handler, enters and deletes most of the information stored in the active tool table.

The tool table handler is called to add entries to the table when MSG OpenConn primitives that are part of RUNTOOL scenarios complete, and to delete entries from it when relevant MSG CloseConn primitives complete. On receipt of an FE-LND-SAVED from the WM, the tool table handler sets the "lnd saved" flag in the connection state vector of the tool table entry.

After it creates a new tool table entry, the tool table handler requests the tool process (via the user process) to begin the tool (see Section 6.4.3). For tool termination, a request is sent to the tool process (via the user process) to finish the tool (see Section 6.4.3). An acknowledgement from the tool process for a finish request is required before the tool table handler is called to delete the entry from the active tool table.

6.5.2 The User Interface Process

In exec mode the driving events for the user interface process are user terminal activity, input from the protocol process, and input from the tool process. In tool mode terminal input is disabled as a driving event for the user process, but input from the tool and protocol processes remains.

In either mode, when input from any driving event occurs, the user process executes the following main loop, regardless of the event. If there is terminal input, the terminal interface module processes it (see Section 6.5.2.1). If there is input from the protocol process, the protocol process interface module processes it (see Section 6.5.2.3). If there is input from the tool process, the tool process interface module is invoked (see Section 6.5.2.4). Finally, the procedure Notify-User is called to output any messages to the user that are ready to be printed as the result of the event(s) that occurred (see Section 6.5.2.2).

The user interface process is built from the following major functional entities: terminal interface module, user interface

manager, protocol process interface manager, and tool process interface manager.

6.5.2.1 Terminal Interface Module

The terminal interface module is the direct FE interface to the user's terminal. When the FE is in exec mode terminal input may occur in several contexts: command context, HELP reply context, and query reply context. The terminal context may be viewed in pa[illegible] determining an input grammar for the FE; that is, depending upon the context, the FE will expect different kinds of input from the user. In command context, terminal input is interpreted as input to form a NSW command. In HELP reply context, terminal input is interpreted as a user response to a HELP query, and in query reply context, it is interpreted as a user reply to a FE query for a command parameter. Terminal output resulting from command execution (e.g., the output signalling the user that a DELETE command has completed) is done when the terminal context is "free".

The terminal interface module includes a collection of routines which implements those features of the user interface common to all terminal contexts and three procedures which implement terminal input processing actions specific to one of the terminal input contexts.

The collection of routines adds characters to and deletes characters from the terminal input buffer. It implements the command-editing functions of the FE, as described in Section 5.3. It also supports the exec mode quote character, user signal, and

"flush output" functions described in Section 5.7. The "flush output" function involves interaction with the user interface manager and with the terminal output buffer manager.

The following describes the procedures which implement the three terminal input contexts.

- Command-Parse

  This procedure implements terminal input processing in command context. It seeks to accumulate a complete NSW command. To do so, it must reference the grammar tree, which stores the grammar in a tabular form. Command-Parse, in effect, determines whether the contents of the terminal input buffer constitute a valid sentence of the command language, a valid but incomplete sentence of the command language, an ambiguous sentence of the command language, or an invalid sentence of the command language. As it recognizes syntactic items (e.g., <command-verb>s, <param>s) it places tokens for them into the parse tree along with pointers to the locations of the corresponding user typein in the terminal input buffer. It modifies the parse tree in response to command-editing functions which change the contents of the terminal input buffer.

  The general terminal interface calls Command-Parse to implement command-recognition and completion, options typeout, and command prompting. The prompt strings which the user may request are stored in the grammar tree. The FE parsing rules described in Section 5.5 are implemented by procedure Command-Parse in conjunction with the general terminal interface.

  When a complete command is detected, the procedure Command-Dispatch is called and appropriate action is taken.

- Help-Reply

  This procedure processes user replies to HELP calls from the WM. These user replies are entered in Help reply terminal context. The help code received by the FE in the HELP call sent by the WM indicates the kind of response expected (i.e., type of data to be supplied by the user) and is supplied to to Help-Reply as a parameter. After the reply has been gathered, it is sent to the protocol process.

- Query-Input

This procedure supports command entry in the query mode. The procedure Query-Input tries to accumulate a syntactically correct command by querying the user for each element of the command which has not been entered by the user. The procedure generates its queries on the basis of information in the grammar tree.

It is not expected that query reply context will be used to enter a <command-verb> or a <qual-item>; rather, its use is expected to be in entering command parameters. For example, query reply context will be supported for entering long commands, such as the "NET" commands. The commands for which the context is supported are limited in number, i.e, it is not intended to be a widely-used means of command entry.

If a complete command has been accumulated, the Command-Dispatch procedure is called.

6.5.2.2 User Interface Manager

The user interface manager is a collection of procedures which implement various functions in the user process. These procedures are called by the user process as it executes its main loop, as well as by by the terminal interface module, by the protocol process interface manager, and by other procedures in the user interface manager.

The procedures are:

- Notify-User

  This procedure is called by the main loop of the user process to inform the user that a command has completed or that a command requires further user interaction before it can be completed; e.g., the WM has made a HELP call on the FE. Completed commands may result in the output of a text string, e.g., SHOW FILES. All text strings received from remote NSW processes are buffered in the user process before being displayed to the user.

  The procedure examines the terminal output buffers and notifies the user each time a new instance of terminal activity is required. It uses the FE-assigned command number, which is stored in the active session table, in its notification. For example, if a DELETE command has been

assigned to be command number 4 and if the command requires further user assistance, the procedure will later output "Help for Delete [4] needed".

- Command-Dispatch

When a <Complete-command> has been detected by Command-Parse or Query-Input, it is necessary to determine the action to be taken. A complete command may require the initiation of a NSW protocol scenario; it may require that the FE perform a task "locally"; or it may require that some action requiring communication with the tool process be taken. The procedure Command-Dispatch determines the action called for and then calls an appropriate procedure. If an NSW protocol scenario is to be initiated, procedure Send-ProtocolProcess is called; if the complete command in the parse tree specifies an action to be taken by the FE, the procedure Execute-Local is invoked; and if an action related to tool-handling is called for, the procedure Send-ToolProcess is called.

- Execute-Local

A "local command" is one which can be performed locally by the FE without the assistance of other NSW system components. Examples are the HELP and DESCRIBE commands, which invoke termnal output of specified help strings to assist the user. Another kind of local command is a user request to display a buffered string associated with a previously issued command. Execute-Local implements this capability and is called by the Command-Dispatch procedure.

- Send-ProtocolProcess

This procedure assembles MSG messages for transmission by the protocol process. It is called with an argument specifying whether the MSG message is to be an initial call to the WM or a HELP call reply.

Send-ProtocolProcess assembles the appropriate MSG message in NSWTP format and assigns a scenario identifier. It computes the message length of the message and determines what command sequencing, if any, is required with respect to existing scenarios. The MSG message, message length, scenario identifier, and sequencing information are then sent to the protocol process. For an initial call to the WM, a new entry is also made in the active session table.

- Send-ToolProcess

This procedure handles the communication of the user process with the tool process. It is called by procedure Command-Dispatch.

If the interaction with the tool process involves no change of context to tool mode, it assembles the message to the tool process and sends it; an example of such a message is a request to set an echo option. If the interaction with the tool process involves a change in context to tool mode, the procedure displays a tool-initiation or tool-resumption text string to the user, disables terminal I/O in the user process, and performs all FE change-of-context protocols with the tool process and the protocol process. An example involving change of context is the REUSE command.

6.5.2.3 Protocol Process Interface Manager

The protocol process interface manager handles pipe communication with the protocol process. It uses the active session table in its operation. For example, it may set the command status field for an entry in the table to indicate that a command that requires the intiation of a protocol scenario has been acknowledged by the protocol process. When the user process learns that a protocol scenario has completed, the protocol process interface manager will request the protocol process to clear its tables of all entries for the protocol scenario. When the protocol process sends a message that requests a string to be displayed on the user's terminal, the protocol process interface manager will buffer the string for subsequent output by Notify-User.

The protocol process interface manager also handles the mediation of tool-initiation and tool-termination messages (for beginning and finishing tools) between the protocol process and the tool process.

6.5.2.4 Tool Process Interface Manager

The tool process interface manager handles pipe communication with the tool process. Three forms of input are expected: a "tool finished" message, a "suspend tool" message, and acknowledgments to previous messages from the user process.

The "tool finished" message acknowledges a request from the protocol process to perform all functions in the tool process needed to terminate a tool. In this case the tool process interface manager sends a message to the protocol process, reenables the terminal for the user process, and signals a return to exec mode.

The "suspend tool" message is received after the user has typed control-N in tool mode. It causes the terminal to be reenabled for the user process and a signals a return to exec mode.

6.5.3 The Tool Interface Process

The events which drive the tool interface process differ slightly between exec and tool modes.

In exec mode, the tool process responds to input from any tool connection which may exist and to pipe input from the user process. Since the user is not directly communicating with a tool in exec mode, the tool process does no terminal I/O. Consequently, no data is sent to tools over tool connections, and any tool connection activity that occurs will be output from the tools. The tool process will buffer output from the tool connections and handle any TELNET protocol obligations.

Pipe input from the user process triggers other activity. The user process may instruct the tool process to begin a new tool, finish a tool, activate an existing tool connection, or set an echo option. Beginning and finishing tools do not open and close tool connections (since those operations are performed in the protocol process), but invoke tool-related data management functions. Beginning new tools and activating suspended ones involve a change in context from exec mode to tool mode.

In tool mode, the user terminal becomes an additional source of input for the tool process. Most user typein is sent directly to the tool. Control-N causes a change in context to exec mode and hence disables further terminal input to the tool process. A limited set of TELNET commands may be entered by the user in tool mode by typing specified characters. Output from tools other than the one in use is buffered as in exec mode.

The tool interface process is built from the following major functional entities: TELNET connection handler, tool mode command processor, and pipe interface module.

6.5.3.1 TELNET Connection Handler

The TELNET connection handler does I/O on TELNET connections to tools and performs certain tool-related data management functions. The tool process interface to TELNET connections is provided by a collection of UNIX TELNET library routines (see Section 7 and appendix A for details).

The TELNET connection handler is a set of procedures, each of which implements a tool connection handling function; these

functions typically involve calls upon the UNIX TELNET library.

The procedures include:

- Begin-Connection

  In the NSW scenario to start a tool, the Foreman instructs the FE to open a TELNET connection to the tool. MSG returns the FE a UNIX file descriptor as the handle for the opened connection. This file descriptor is the argument to procedure Begin-Connection.

  Begin-Connection allocates a tool buffer for the new tool and enables the user terminal and TELNET connection for I/O. Following completion of Begin-Connection, the TELNET connection to the tool is ready for use for tool communication. The internal FE synchronization protocol assumes that control of the user's terminal may be taken by the tool process upon invocation of Begin-Connection. Begin-Connection also sets the "connection in use" flag in the active tool table entry for the tool and sets the "current tool" pointer to the active tool table entry for the tool.

- Read-Connection

  Read-Connection reads from a specified tool connection. If the user is currently using the connection (i.e., if "connection in use" is set), the data read from the connection is output directly to the user's terminal. If the user is not currently using the connection, the data from the connection is buffered, the buffer pointer and character count are appropriately updated, and the "output waiting" flag is set in the tool's entry in the active tool table.

- Write-Connection

  Write-Connection writes into a specified tool connection.

- Send-Command

  By typing specified control characters the user may cause certain TELNET commands to be written into the connection. "Erase Line" and "Erase Character" are two examples. Send-Command implements this capability.

- Suspend-Connection

  This procedure "suspends" but does not close a specified tool connection. When Suspend-Connection is called, the tool process turns off the "connection in use" flag in the

active tool table entry for the tool and returns control of the user's terminal to the user process. Suspend-Connection is invoked when the user types control-N in tool mode.

- Activate-Connection

  Activate-Connection reactivates a specified suspended connection. It sets the "connection in use" flag in the tool's active tool table entry. The FE internal synchronization protocol assumes that control of the user's terminal may be taken by the tool process. If output has been buffered for the tool connection, the user is informed, is given a character count of the buffered data, and is given opportunity to either flush the buffer or type out the buffer.

- Finish-Connection

  The termination of a tool session is handled by the protocol process. The protocol process, having engaged in the tool-halt scenario, i.e., FM-ENDTOOL followed by FE-TOOLHALTED or FE-TOOLHALTED followed by FM-ENDTOOL, then issues a MSG CloseConn primitive to close the direct network connection.

  After the connection has been closed Finish-Connection is invoked by a message from the protocol process (through the user process). It deallocates the tool buffer associated with the tool and clears all tool buffer information. When complete, Finish-Connection responds with a message to the protocol process (through the user process), confirming that the tool session has been finished by the tool process. The protocol process then clears the corresponding active tool table entries.

- Negotiate-Option

  TELNET option negotiation is performed by this procedure. In the initial implementation, only local/remote echo option negotiation (supported in exec mode by a command) will be supported.

6.5.3.2 Tool Mode Command Processor

The tool mode command processor handles terminal I/O in tool mode. Terminal input in tool mode falls into the following classes:

- Control-K.
  Typing control-K in tool mode causes the tool buffer for the connection in use to be flushed.

- Control-N.
  Typing control-N in tool mode causes the procedure Suspend-Connection to be invoked. Control of the user terminal is transfered to the user process.

- TELNET Command Control Characters.
  A predefined set of control characters are defined in tool mode to cause TELNET commands to be sent over the connection currently in use. Typing any of these characters invokes procedure Send-Command.

- Tool Quote Character.
  Typing the tool quote character in tool mode causes the character following it to be sent to procedure Write-Connection without performing any special action the character might otherwise invoke.

- All Other Characters.
  All remaining characters in tool mode are treated as tool input and are conveyed directly to Write-Connection. For alphabetic characters, no case conversion is performed.

6.5.3.3 Pipe Interface Module

The pipe interface module handles pipe transmissions from the user process to the tool process. When a transmission is received, the pipe interface module determines an appropriate procedure to call.

7. Changes to UNIX

We believe that changes to UNIX system software in the areas of terminal handling, user TELNET support, and the NCP should be made to support the UNIX NSW Front End. This section describes the changes required. Familiarlity with the UNIX operating system is assumed.

7.1 Terminal Handler Modifications

The UNIX terminal handler must be modified to give the following capabilities:

a. User can set or unset any of 16 break character classes; the break classes are predefined and cannot be changed by user.

b. User can specify that the break character should or should not be echoed.

c. User can specify that the text up to, but not including, the break character should or should not be echoed.

d. User can specify that echoing should occur at the time the character is read by the user program ('deferred echoing'), or when the character is received by the terminal handler.

e. User can override the UNIX convention of throwing away terminal input to a full terminal buffer (256 characters); the capability to instruct the terminal handler to reject further terminal input until buffer is cleared will be provided (a warning character, e.g., <bell>, will be echoed).

Capabilities (a), (b), and (c) are specified in the TELNET Remote Controlled Transmission and Echoing (RCTE) option (NIC 19859). The first nine break classes will be those specified in the RCTE specification. The remaining seven classes are to be determined according to UNIX and NSW needs.

These capabilites will be implemented by means of a new UNIX system call ttymod, having the following C language interface:

ttymod(fd,get/set,op,ptr,len),

where:

fd	is file descriptor of terminal,
get/set	is 0 or 1, respectively,
op	is code to indicate operation,
ptr	is pointer to user parameter block,
len	is length of user parameter block.

The ttymod system call will incorporate all capabilities present in the current UNIX stty call, in addition to the capabilities stated above.

7.2 TELNET Program Interface

Implementation of user TELNET functions of the UNIX FE will be facilitated by utilizing TELNET library routines which are called to perform all TELNET I/O and protocol handling. The specifications for the library routines are given in Appendix A.

In addition to the routines specified in the appendix, we suggest that the following three additional functions are to be added to the TELNET library routines:

- telinit

  Given an already open connection, telinit initializes the TELNET library routines to work on this connection.

  Synopsis:

  result = telinit(fd)

  where:

fd	is the file descriptor of the open connection;
result	is a success (0) or failure (-1) flag returned.

- telfinish

This function does the inverse of telinit, i.e., it clears all tables and entries used by the TELNET library routines for the connection, but does not close the connection.

Synopsis:

```
        result = telfinish(fd)
where:
fd      is the file descriptor of the connection;
result  is a success (0) or failure (-1) flag returned.
```

- telstatus

This function is used to determine if a given TELNET option is in effect on a given connection or not.

Synopsis:

```
        result = telstatus(fd,option)
where:
fd      is the file descriptor of the connection;
option  is the option whose status is desired;
result  is a flag that the option is in effect (1) or
        is not in effect (0).
```

7.3 UNIX NCP Modifications

The NSW FE design calls for all TELNET handling to be done by one process. However, at present no user process on UNIX can support more than one open TELNET connection-pair at a time.

The problem is the following. Host-to-host protocol interrupts (INS and INR) are processed by the UNIX NCP and are forwarded by means of a signal (software interrupt) to the user TELNET process. If the user TELNET process has more than one connection open at the time a host-to-host interrupt arrives, the user process will be unable to determine which connection is responsible for the interrupt.

The solution we propose is the following. The UNIX NCP will maintain a count of the number of INSs and INRs received on each

connection; this count will be kept in the socket fields of the open parameter block for each special network file. It will be the responsibility of the user program to explicitly ignore signals.

The UNIX system call fstat will be modified so that it can return the INS and INR counts maintained in the socket fields and then zero these fields.

Another UNIX NCP problem exists which has prevented the implementation of the MSG OpenConn and CloseConn operations. These operations are required by the FE in order to establish connections with NSW tools. The implementation of these primitives requires that MSG be able to determine ARPANET socket numbers in advance of opening the connection. These socket numbers must be communicated by the local MSG to the remote MSG prior to opening the connection.

The UNIX NCP currently does not provide any way for a process to determine in advance the socket number to be used for a connection, and must therefore be modified to do so.

The solution we propose is the following. Add a new open type bit to the open parameter block for the connection that means "do a dummy open". If used with a zero local socket number, this will result in a block of eight sockets being reserved and a file descriptor being returned that can be used for relative opens. The file descriptor returned is associated with the base socket of the block of eight. This base socket number can be determined using the UNIX system call fstat. MSG

can thus determine what local socket number will be used when it does a relative open and can send this information to the remote MSG before doing the open.

The socket associated with the dummy open file descriptor will be treated as if it were in a special pre-open state, similar to the listening state except that any incoming RFC matching the socket number will not cause any change of state, but will be treated as any other unmatched RFC. Other operations using the file descriptor will either hang, do nothing, or cause an error as appropriate.

The block of eight sockets will remain allocated to the process until all associated file descriptors have been closed (or the process dies). This is consistent with current practice. Note that when other file descriptors are associated with the block, it will be possible to close the dummy file descriptor and reopen it (with offset zero) as an actual connection.

If an open occurs with the dummy bit set and with a local socket specified, the NCP will essentially act as it does currently. If the socket number is greater than or equal to 1024, it will do nothing and give an error indication. If it is less than 1024, it will return a file descriptor associated with that specified socket (which is in the dummy open state). The user process can then presumably use that file descriptor to do relative opens. No more checking for conflicts with other processes will be done than is presently done.

8. Proposed Change to NSWTP

There are two situations which arise in currently defined NSW protocol scenarios that indicate a fundamental deficiency in the NSW Transaction Protocol (NSWTP). These situations arise in connection with tool-startup and with the processing of HELP messages.

- Tool Startup

  In NSW RUNTOOL scenarios after the Works Manager has arranged with a Foreman to start a tool, the Foreman instructs the FE to open a connection to it to support tool communication. The Foreman makes the request by sending the FE an FE-OPENCONN message. The message has a null NSWTP transaction identifier (since FE-OPENCONN does not require a reply) and its NSWTP type field indicates that the message is an "initial call". The FE-OPENCONN message by itself does not provide the FE with enough information to identify the RUNTOOL scenario, if any, it corresponds to. That is, if the FE has multiple RUNTOOL scenarios in progress, it is not possible for it to determine the one that the FE-OPENCONN message corresponds to. After the FE receives a reply from the Works Manager it can match the FE-OPENCONN message with the correct scenario.

- HELP calls

  In certain NSW scenarios the FE may be queried for "help" in order to permit the scenario to proceed to completion. Typically these HELP calls come from the Works Manager and involve file operations. A HELP call message contains an NSWTP transaction identifier selected by the process requesting help (e.g., a Works Manager) and a type field that indicates that it is an "initial call". There is nothing in the message itself that enables the FE to determine the NSW scenario it corresponds to. Unlike the RUNTOOL situation described above, the NSWTP transaction identifier is not null. However, it is assigned by the requesting process, and cannot be related to a previously FE assigned transaction identifier for the message that initiated the scenario requiring help.

Solutions to both the tool-startup and HELP call problems have been devised.

- Tool Startup

  One approach is for the FE to defer attempting to match FE-OPENCONN messages with protocol scenarios until the corresponding reply from the Works Manager is received. The FE can, of course, open the tool connection requested when the FE-OPENCONN message arrives, but it must defer associating the connection with an entry in the active tool table.

  Alternatively, the FE may impose upon itself the constraint that at most one RUNTOOL scenario be in progress at a time. In this case any FE-OPENCONN corresponds to the only active scenario.

- HELP calls

  One solution to the problem of matching HELP messages with on-going scenarios is to limit the context of FE operations in a way that makes it possible to deduce the proper scenario for a HELP call message. This may be done by imposing the restriction that the FE have at most one HELP-call producing scenario in progress at any time. This is done in the current TENEX FE.

  Alternatively, the HELP call protocol could be modified to provide the information necessary to match calls with scenarios, for example as recently proposed by Kirk Sattley of Massachusetts Computer Associates.

In our view these problems reflect the same underlying deficiency in NSWTP. We believe that this deficiency, even if it can be overcome in specific situations as described above, is likely to reappear in different contexts as new protocol scenarios are defined. The basic deficiency is that NSWTP does not support the notion of system-wide scenario identifiers. The current use of transaction identifiers is adequate for simple protocol scenarios, such as send-receive interactions involving only two NSW components, but proves inadequate for more complicated scenarios.

AD-A084 088 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA F/G 9/2
UNIX NSW FRONT END.(U)
MAR 80 R H THOMAS F30602-78-C-0242
UNCLASSIFIED RADC-TR-80-44 NL

END
DATE FILMED
6 80
DTIC

A system-wide scenario identifier would enable any NSWTP message to be matched with the on-going protocol scenario to which it belongs based only on the message itself. The scenario identifier would be assigned by the NSW component initiating the scenario and would be included in a scenario identifier field in all NSWTP messages sent by any NSW component participating in the scenario. The principal requirement for scenario identifiers is that they be unique; i.e., no component should use the same identifier for two active scenarios, and no two components should ever generate the same identifier.

In this document we propose no specific implementation of scenario identifiers within NSWTP. Our objective is merely to suggest that they would be very helpful in simplifying the processing of messages in several situations, and in increasing the flexibility with which system components can participate in protocol scenarios.

9. UNIX MSG Enhancements

The implementation of MSG on UNIX is incomplete and will currently not support all required FE functionality. In particular, the MSG OpenConn and CloseConn operations have not been implemented yet. These MSG operations will be implemented as part of the UNIX FE implementation effort. Their implementation depends upon the UNIX NCP modifications described in Section 7.3.

10. Support for Management Tools

NSW permits project managers to create and delete project nodes, and to assign access rights to and remove them from project nodes. These functions are supported by a collection of tools known as "management" tools. The management tools currently supported are create-node, delete-node, assign-rights, and remove-rights.

In the current NSW system the management tools are actually implemented largely within the TENEX FE. That is, although the management tools are invoked by a "use" command, and although it appears to the user that such a tool is executing on a tool bearing host, in fact the FE recognizes the tool name as the name of a management tool and interacts directly with the user as if it were the tool, rather than initiating a RUNTOOL scenario with the Works Manager to start the tool. The FE interacts with the user to gather the information required, and then issues a call upon a Works Manager procedure to perform the management function by sending the Works Manager a generically addressed message.

The issue here is not whether this implementation approach for management tools could be supported by the UNIX FE. It could be in a straight forward manner. The query mode input mechanism described in Sections 5.7 and 6.5.2.1 could be used within the UNIX FE to support the currently defined NSW management tools. The FE would enter query mode when it recognized a management tool name and would behave like the TENEX FE from that point.

The issue here is what the proper implementaion approach for NSW management tools should be: should they be implemented within NSW FE's or is there a better implementation approach.

Two major problems with the FE implementation approach are:

- Every full function FE must support management tools. New FE's must, in effect, reimplement all the management functions.
- As new management functions are specified all full function FE's will have to be modified to support them.

We believe that management tools should be implemented either as standard tools that run on tool bearing hosts, or as "pseudo" tools that are implemented by some NSW system component (e.g., the Works Manager or some new component). Regardless of the approach selected, the FE should be able to treat management tools in the same way it treats any other tool. For example, to start a management tool it would engage in a RUNTOOL scenario with the Works Manager to establish communication with the tool (or the system component that implements the "pseudo" tool).

We believe that the issue of management tool implementation is an important one and that it must be settled before NSW is released for operational use. Our current plan for dealing with management tools within the UNIX FE implementation effort is to defer implementation of the management tool functionality within the FE in the expectation that the issue will be resolved and that FE's will not be required to provide special support for management functions. If the issue is not resolved as our implementation effort nears completion, we will provide

management tool support within the UNIX FE by means of the query mode input and flexible protocol scenario definition mechanisms.

11. References

1. R. Millstein, "The National Software Works: A Distributed Processing System", Proceedings 1977 Annual ACM Conference, Oct 1977.

2. H. Forsdick, R. Schantz, R. Thomas, "Operating Systems for Computer Networks", IEEE Computer Magazine, Jan 1978.

3. S. Schaffner, S. Sluizer, "Works Manager Subsystem Specificiation", Massachusetts Computer Associates Report No. CADD-7709-2712, Nov 1978 revision.

4. R. Schantz, R. Millstein, "The Foreman: Providing the Program Execution Environment for the National Software Works System", BBN Report No. 3442, Massachusetts Computer Associates Report No. CADD-7701-011, Jan 1977.

5. P. Cashman, R. Faneuf, C. Muntz, "File Package: The File Handling Facility for the National Software Works", Massachusetts Computer Associates Report No. CADD-7612-2411, Dec 1976.

6. R. Thomas, "MSG System/Subsystem Specification", BBN Report, Aug 1978.

7. R. Millstein, R. Shapiro, "Interim NSW Reliability Plan", Massachusetts Computer Associates Report No. CA-7701-2111, Mar 1977 revision.

8. Massachusetts Computer Associates, "Third Semi-annual Technical Report for the National Software Works", Massachusetts Computer Associates Report No. CADD-7702-2811, Feb 1977.

9. R. Rom, "Minimum Front End Specification", SRI Report, NIC No. 28672, Sept 1976.

Appendix A: Specification of TELNET Library Subroutines

A. TELNET Library Routines

The following routines are intended to make the task of interfacing to a TELNET connection simpler. They eliminate the need for the programmer to be aware of such details as socket numbers and the particular type of open required. They also reduce the need for the programmer to know about the low-level aspects of the protocol -- e.g., the actual numeric representation of DO, WILL, WONT, DONT, and individual options.

A.1. Telopen

The telopen routine is used to open a standard TELNET connection to a specified host. The programmer must know the number of the host on the network. Once the connection has been opened, the routines telwrite and telread should be used to transmit and receive over the connection.

Synopsis:

```
fd = telopen(hostnum);
```

where:

hostnum — is the numeric address of the host;

fd — (returned) is the file descriptor for the newly opened connection, or -1 if the connection could not be opened.

Note: this routine sets up a handler for the NCP signal SIGINR. If the signal occurs during a system call, the system call may abort prematurely. Any system call which can be aborted by a signal (for example, reading or writing a terminal should be checked to see if it returned -1. If it did, 'errno' should be checked to see if it is equal to 4 (EINTR); if so, the system call was interrupted by a signal and the appropriate action should be taken. An appropriate action might be retrying the system call, if it was a read, or merely giving up on it, if it was a write. See the description of SIGNAL in section II of the UNIX manual.

A.2. Telread

The telread routine is called to read from the file descriptor returned by telopen. It performs a read system call on the file descriptor, and scans for TELNET commands. If any are found, it takes the appropriate action, which usually includes writing a reply back. It returns the remaining data.

Synopsis:

```
nactual = telread(fd, buffer, nreq);
```

where:

fd — is the file descriptor returned from a call to telopen();

buffer — is a pointer to a buffer to be filled in by telread;

nreq — is the number of bytes to be read;

nactual — (returned) is the actual number of bytes read. This may be fewer than the number of bytes requested. If it is 0 (EOF), the foreign host has closed the connection. If -1, some error has occurred. As with the system read and write calls, the external variable 'errno' has been set to indicate the error. Under most circumstances, the program should assume that the connection has been closed.

This routine performs a system read call to read the requested number of bytes into the user's buffer. It then scans the buffer for 0377 octal, which flags a TELNET command. If any are found, it processes the command, possibly writing data onto the file descriptor, and then deletes the command from the buffer. When it has finished processing all commands, it will return the remaining data to the user.

A.3. Telwrite

The telwrite routine is called to write onto the file descriptor returned by telopen. Its sole task is to prevent user data from appearing to be TELNET commands, by scanning for bytes with a value of 0377 octal. If any are found, they are sent doubled, as specified by the protocol. The TELNET software on the foreign host will pass a single 0377 byte to the user program for each such pair of bytes it sees.

Synopsis:

```
nactual = telwrite(fd, buffer, nrequested);
```

where:

fd — is the file descriptor obtained from the telopen call;

buffer — is a pointer to the data to be written;

nreq — is the number of bytes to be written;

nactual — (returned) is the actual number of bytes written. It should be regarded as an error if this is not the same as the number of bytes requested.

A.4. Telclose

The telclose routine is called to close the TELNET connection. The termination of a process closes the TELNET connection automatically; this routine is provided for those circumstances where it is desired to close the connection and continue the process.

Synopsis:

```
telclose(fd);
```

where:

fd — is the file descriptor returned from the telopen call.

A.5. Setopt

The setopt routine provides the user control over the option negotiation process carried out by telread. It permits the user to specify what option requests presented by the foreign host are to be accepted. By default, all option requests will be refused.

Synopsis:

```
oldaction = setopt(option, action);
```

where:

option — is the number of the option whose action is to be changed.

action is 0 to cause this option to be negatively acknowledged; 1 to cause this option to be positively acknowledged, but otherwise ignored; or the address of a function to be called when the option is negotiated (see below);

oldaction (returned) is the action previously set for this option, or -1 if an action cannot be set.

If a function is specified, it will be called as follows:

```
ack = function(fd, flag, option);
```

where:

fd is the file descriptor of the connection which requested the option;

flag is either 0, 1, 2, or 3 depending on whether the foreign host sent a WILL, DO, WONT, or DONT respectively;

option is the option requested; this permits the same routine to handle more than one option;

ack (returned by the function) should be 0 if the option should be negatively acknowledged, or any nonzero value if the option should be positively acknowledged.

A.6. Sendopt

The sendopt routine is used to send option requests over a TELNET connection. A program wishing to know whether the request was successful should first call setopt to set up a handler for the option. When the handler is called for a reply, its returned value is ignored.

Synopsis:

```
error = sendopt(fd, flag, option);
```

where:

fd is the file descriptor returned by the telopen call;

flag is either 0, 1, 2, or 3 to send a WILL, DO, WONT, or DONT respectively;

option is the option requested;

error is 0 if there was no error, -1 if an error occurred.